Performance Optimization of an Order Book

Group 14

Abstract

This report compares a naïve order book implementation with an optimized version. Using insert/amend/delete results, we compare data structures, complexity, scaling behavior, where the naïve design fails, and why the optimized design scales.

1. What data structures you chose and why

Naïve implementation:

- bids[] and asks[] stored as lists and kept sorted by price.
- Order lookup by order_id requires scanning lists.
- After updates, lists are resorted to restore price order.

Optimized implementation:

- orders_by_id: dict(order_id -> Order) for O(1) average lookup on amend/delete.
- price_levels: dict((side, price) -> dict(order_id -> Order)) to update within a price bucket.
- Heaps over active price levels per side for best bid/ask, which is lazy cleanup of empty levels.

2. Complexity analysis of each operation

Let n be the number of active orders and m the number of distinct price levels.

Operation	Naïve (lists + resort)	Optimized (dicts + levels + heap)
Insert	O(n log n) (append + sort)	O(1) avg book keeping; O(log m) if a new price level is added
Amend	O(n) find + O(n log n) resort	O(1) avg lookup/update; O(log m) only if price changes level
Delete	O(n) find + O(n log n) resort	O(1) avg lookup/delete; heap cleanup amortized

The baseline repeatedly takes global sorting cost; the optimized version performs localized constant-time updates and only minor heap maintenance.

3. Observed scaling behavior

3.1 Total Time

Figure 1. Total runtime for insert/amend/delete (naïve vs optimized)

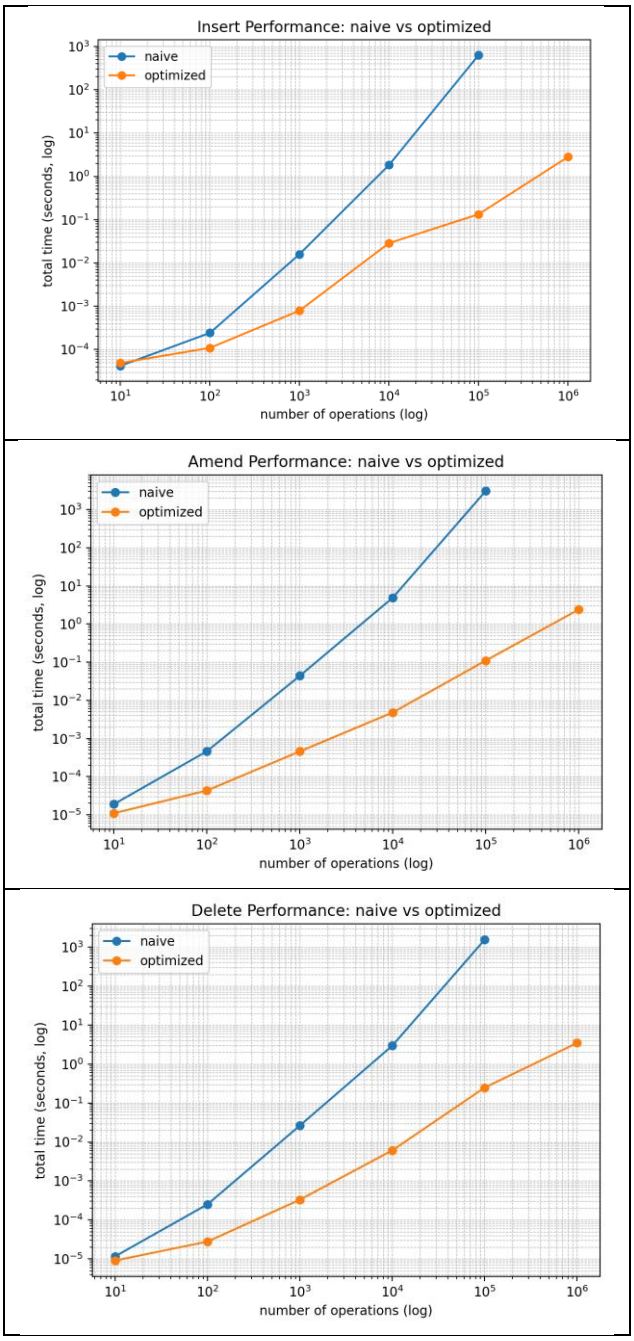


Figure 1 plots total runtime vs workload size n. Insert/amend/delete in the naïve version grow much faster comparing to the optimized version with n. Since it takes too long for naïve version to finish the benchmark, I did not show the naïve result for n=1,000,000.

3.2 Average Time

Operation	Method	n = 10^1	n = 10^2	n = 10^3	n = 10^4	n = 10^5	n = 10^6
Insert	Naïve	4.20e-06	2.42e-06	1.59e-05	1.82e-04	6.28e-03	N/A
Insert	Optimized	4.89e-06	1.09e-06	7.90e-07	2.86e-06	1.33e-06	2.82e-06
Amend	Naïve	1.90e-06	4.61e-06	4.39e-05	4.92e-04	3.07e-02	N/A
Amend	Optimized	1.10e-06	4.32e-07	4.58e-07	4.82e-07	1.10e-06	2.39e-06
Delete	Naïve	1.17e-06	2.52e-06	2.65e-05	2.95e-04	1.54e-02	N/A
Delete	Optimized	9.10e-07	2.79e-07	3.31e-07	6.10e-07	2.48e-06	3.45e-06

Naïve implementation: As n increases, the average time per operation rises noticeably, especially for amend and delete. For example, amend increases from 4.39e-05 at n = 10^3 to 3.07e-02 at n = 10^5 (~699.9×). Delete increases from 2.65e-05 at n = 10^3 to 1.54e-02 at n = 10^5 (~581.7×).

Optimized implementation: Avg time per operation is much more stable, with far smaller growth than the naïve version. For amend, avg_sec changes from 4.58e-07 at n = 10^3 to 2.39e-06 at n = 10^6 (~5.2×).

4. Where the naïve version breaks down

The baseline becomes impractical when n > 10,000, because amend and delete combine an O(n) search with an O(nlogn) full resort after each update. The repeated sorting dominates total runtime, producing linear growth as the book size increases.

5. How the optimized version improves performance

The optimized design removes the baseline's bottlenecks:

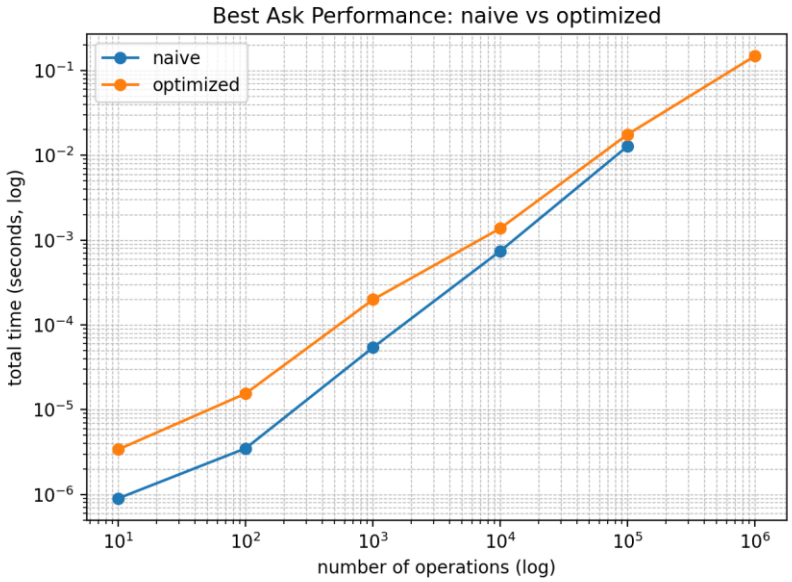
- O(1) average lookup via orders_by_id eliminates linear scans in amend/delete.
- Price level buckets localize updates and avoid full list resorts after each operation.
- Heaps provide fast best bid/ask without maintaining a globally sorted list.

Speedup summary (naïve time / optimized time) over benchmarked n values:

Operation	Min	Median	Max
insert	0.9x	20x	4728x
amend	1.7x	96x	27894x
delete	1.3x	80x	6213x

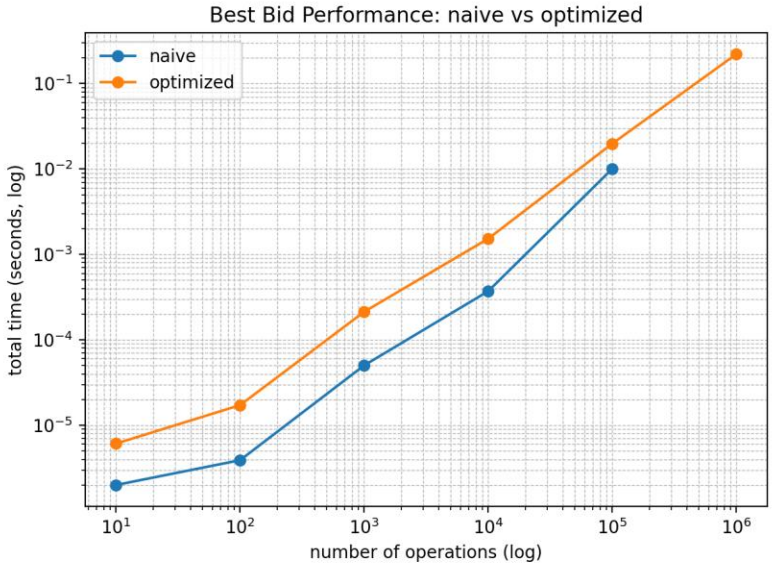
Appendix

Figure A1. Best ask query performance



The optimized version is slower for best-ask queries across the tested range. This is consistent with higher constant overhead from heap-based access and lazy cleanup logic. In the naïve version, best ask can be returned by directly reading the first element of an already sorted asks list.

Figure A2. Best bid query performance

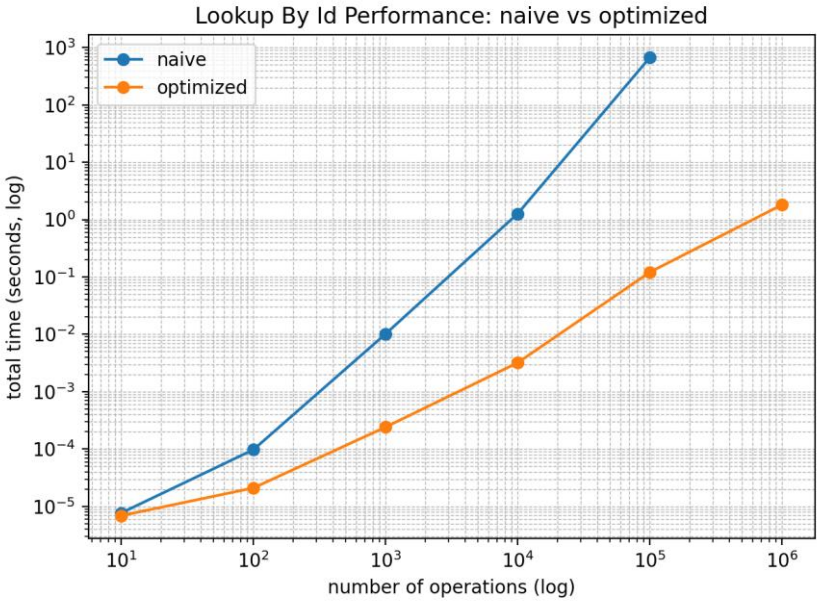


Similar to best ask, the optimized version is slower for best-bid queries, suggesting the same constant-factor overhead dominates this benchmark.

Figure A3. Get-orders-at-price performance

In the naïve implementation, retrieving orders at a given price typically requires scanning through all orders on that side (O(n) per query). Repeating this n times leads to near O(n^2) total behavior. In the optimized implementation, price_levels provides direct access to a price bucket (O(1) average to locate the bucket, plus iterating only the orders at that level). This avoids scanning unrelated orders.

Figure A4. Lookup-by-id performance



Naïve lookup by order_id requires a linear scan across lists (O(n) per lookup). When repeated n times, this produces near O(n^2) total runtime. Optimized lookup uses orders_by_id (dictionary) for O(1) average-time retrieval, so total runtime grows close to linearly with the number of lookups.