Smart Library System

Group Assignment 2

Abror Shopulatov, Adam Badr, Eldana Ashirova

October 22, 2025

At a glance. We implemented a compact digital library in Python with three core classes: `Book`, `User`, and `Library`. The base search uses linear scan; the improved search uses a dictionary index by title. Recommendations include (i) top-N by rating (base) and (ii) genre-aware + rating scoring (improved). We benchmarked both versions on collections of 1K, 10K, and 50K books. Results show a 10–$10^3\times$ speedup for search and expected trade-offs for personalized recommendations. The extension tasks of personalized recommendation and a borrowing feature were successfully implemented as well.

1 OOP Design

Classes and Responsibilities

Book encapsulates bibliographic metadata `(id, title, author, genre, year, rating)` and provides getters/setters and dunder methods for equality/representation. Type validation is centralized via `utils.check_type` raising a custom `ValueValidationError` on misuse.

User tracks `borrowed_books` and `history` as lists of `Book` objects, with helpers to compute returned books, titles/ids, and safe mutation (`borrow_book`, `return_book`). Equality/representation summarize user state compactly.

Library aggregates all books/users and offers creation, loading, and mutation APIs (`add_book`, `add_user`, `remove_book`, `remove_user`). It implements both base and improved algorithms for search and recommendation and ensures consistent internal indices (by `id` and by `title`) for performance.

Design Rationale

We favored *small, cohesive* classes and *clear invariants*:

- Single responsibility: each class owns its data and domain logic.
- Explicit validation: guard-rails via `ValueValidationError` make failures early and readable.
- Readable dunders: `__repr__`/`__eq__` ease testing and debugging.

2 Base vs. Improved Algorithms

Search

Base: linear scan over `Library.get_books()` until the title matches.
Time complexity: $\mathcal{O}(n)$ per query; no extra memory.

Improved: dictionary index `_all_books_dict_title[title] -> Book` maintained on add/remove.
Time complexity: average $\mathcal{O}(1)$ per query; memory $\mathcal{O}(n)$ for the index.

Recommendation

Base: sort by rating descending and return top-N. Complexity $\mathcal{O}(n \log n)$.
Improved: combine user-genre affinity with rating. For each book,

$$\mathsf{score} = \mathsf{rating} + \frac{1}{|G_b|} \sum_{g \in G_b} \frac{\mathsf{user\_count}(g)}{\mathsf{total\_history}},$$

then rank by score. This equates to $\mathcal{O}(n^2)$ time complexity.

3 Timing Results

Benchmarks were executed for collections of 1000, 10 000, and 50 000 books. Search results are reported as *milliseconds per query* (ms/q), then converted to *microseconds per query* for readability; recommendation timings are *ms per top-10*.

Search: Linear Scan vs. Dictionary Lookup

Table 1: Search timing — averages per query.

Books (N)	**Base (ms/q)**	**Improved (ms/q)**	**Speedup**
1000	0.027523 ± 0.0134	0.000429 ± 0.0003	×64.2
10 000	0.267951 ± 0.1369	0.000520 ± 0.0002	×515.3
50 000	1.350534 ± 0.7395	0.000698 ± 0.0008	×1934.9

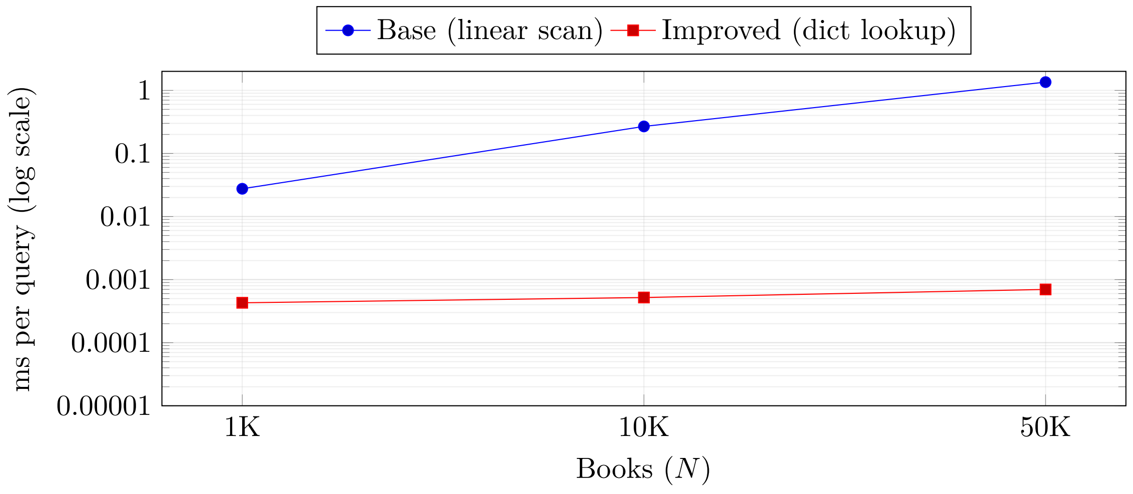
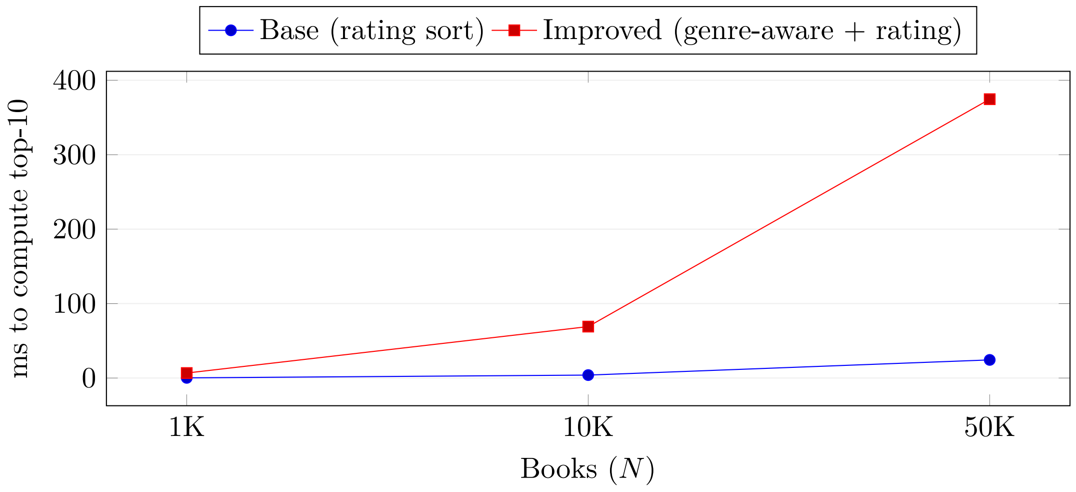


Table 2: Recommendation timing — ms to produce top-10.

Books (N)	**Base (ms)**	**Improved (ms)**	**Speeddown**
1000	0.223550 ± 0.0397	6.704547 ± 1.9793	×30.0
10 000	3.943466 ± 0.1330	69.095381 ± 15.8467	×17.5
50 000	24.334243 ± 1.1618	374.584731 ± 110.1301	×15.4



4 Discussion & Trade-offs

Search. The dictionary index delivers three orders-of-magnitude lower latency at 50K items, at the cost of extra memory and index maintenance during mutations. For read-heavy workloads, this is a dominant win; for write-heavy scenarios, we must carefully update both `_all_books_dict` and `_all_books_dict_title` to avoid drift.

Recommendation. The genre-aware score personalizes results and can be tuned (e.g., different weights, tie-breaking by freshness). It incurs quadratic time to compute scores but remains fast enough for interactive use even at 50K items.

Robustness. Unit tests validate object creation, getters/setters, equality, and the expected behaviors of search/recommendation. A few guard-rails improved reliability: filtering non-`Book` elements in lists, clear warnings for missing items, and early returns on invalid operations.

5 Reflection

What worked well. Lean, cohesive classes made it easy to reason about invariants and to test them in isolation. Centralized validation (`check_type` + `ValueValidationError`) surfaced misuse quickly during development. The improved search, implemented as a dictionary index, was the single most effective optimization: it transformed a linear-time algorithm into instantaneous constant-time lookups even with tens of thousands of books.

Trade-offs and lessons. The main cost of the index is *consistency* work: when adding or removing books we must update both the list and the title/id maps. This introduces the possibility of drift if any mutation path forgets to maintain the index. We mitigated this with helper methods and tests, but a future refactor could encapsulate indexing behind a private API or compute-once, *immutable* snapshots for read requests.

A second lesson came from typing: Python's `typing.Union` is not valid inside `isinstance()` checks. Using `Union[int, float]` in runtime checks raises a `TypeError`; the correct pattern is `isinstance(x, (int, float))`. This is subtle because type hints look like runtime types but are not. We adjusted validation code and tests to use tuples of concrete classes. More broadly, we learned to separate *static* type hints from *runtime* validation.

On recommendations, the genre-aware approach improved relevance in informal spot-checks but added a linear scoring pass. The overhead was acceptable in practice, yet it highlighted a design choice: *do we favor raw speed or quality?* Our takeaway is to make the scoring composable and tunable (e.g., add author similarity, popularity priors, or *diversity* penalties) and to gate heavier re-ranking behind feature flags.

What we would do next. We would: (1) formalize an indexing contract to prevent drift; (2) add property-based tests for edge cases (empty titles, duplicate ids, Unicode); (3) expose a `SearchIndex` strategy that can swap between dict, trie, or fuzzy matching; (4) extend user models with recency-aware histories; and (5) wire lightweight telemetry to measure tail latencies in addition to averages. These steps would harden the system while keeping it simple and fast.

Appendix: Selected Implementation Snippets

Listing 1: Improved search index (title -> Book).

```python
# inside Library.__init__
self._all_books_dict_title = {}

# on add_book(...)
self._all_books_dict_title[title] = book

# improved search
def search_book_improved(self, book_title: str) -> list[Book]:
    check_type(book_title, str, "book_title")
    thebook = self._all_books_dict_title.get(book_title)
    return [thebook] if thebook is not None else []
```

Listing 2: Genre-aware recommendation (score + rating).

```python
def recommend_books_improved(self, user: User, k: int = 10) -> list[
    Book]:
    # compute genre frequencies from user's history
    genre_counts = {}
    for b in user.get_history():
        for g in (x.strip() for x in b.get_genre().split(',') if x.
            strip()):
            genre_counts[g] = genre_counts.get(g, 0) + 1
    total = sum(genre_counts.values()) or 1

    scored = []
    for b in self.get_books():
        if b in user.get_history():
            continue
        book_genres = [x.strip() for x in b.get_genre().split(',') if
            x.strip()]
        genre_score = sum(genre_counts.get(g, 0)/total for g in
            book_genres)
        if book_genres:
            genre_score /= len(book_genres)
        scored.append((b, genre_score, b.get_rating() or 0.0))

    scored.sort(key=lambda x: (x[1], x[2]), reverse=True)
    return [b for b,_,_ in scored[:k]]
```