Collen Foster Ellis
Graham Preston
DAAR
16/11/2025

Assignment 3: Librarian App

Introduction

Problem definition

Modern search engines must handle large volumes of text, provide fast and accurate results, and support meaningful ranking and recommendation features. As the size and complexity of digital libraries continue to grow, efficient indexing strategies and well-designed data structures become essential for supporting real-time interaction. This project addresses these challenges by developing a complete web-based search engine capable of handling a sizable corpus of full-length books and returning relevant results with minimal latency.

The dataset consists of more than 1,664 public-domain books sourced from Project Gutenberg, each containing over 10,000 words. These documents vary widely in vocabulary, writing style, structure, and subject matter, making them a realistic testbed for large-scale information retrieval. Many of the books are old translations, original editions, or texts with complex formatting, which introduce additional irregularities into the tokenization and cleaning process. Because the corpus includes everything from fiction and philosophy to scientific works and historical accounts, the underlying indexing system must be flexible enough to represent very different styles of writing, and robust enough to handle non-uniform formatting, inconsistent punctuation, and vocabulary that spans multiple centuries of English.

Another major consideration is vocabulary size. After cleaning, stop-word removal, and normalization, the corpus still reaches over 600,000 unique tokens, ranging from common terms to highly specific proper nouns. Long documents with this level of lexical diversity require indexing strategies that remain performant even when dealing with tens of millions of characters. A naïve search approach would scan every file linearly for every query which would be too slow to serve as the foundation of a responsive web application. To make the system usable, the search process must rely heavily on precomputation, in-memory indexing, and efficient vector operations.

Technical overview

To meet these needs, the system is implemented as a full-stack web application with a Vue 3 frontend and a Django REST Framework backend. All search logic, text processing, indexing, ranking, and recommendation features occur entirely on the server. The backend loads several precomputed data structures at startup, including a TF-IDF matrix, token sets, vectorization results, and optional inverted indexes, allowing the application to answer queries quickly without reprocessing large text files. On each startup, the backend also checks whether any new documents have been added to the corpus. If updates are detected, the index structures are rebuilt to ensure that the search engine always reflects the most current state of the dataset.

The search engine supports four main user-facing capabilities:

1. Keyword Search
    - Returns documents containing one or more query terms, matched using TF-IDF. This is the primary mode for general queries and is optimized for both single-word and multi-word input.
2. Regex Search
    - Allows users to match lexical patterns using regular expressions applied to the indexed vocabulary. This approach avoids scanning full text files and keeps regex queries fast, even though the underlying documents are large.
3. Two ranking methods
    - A baseline ranking based directly on TF-IDF with cosine similarity.
    - A secondary option that adjusts the ranking using a closeness-style distance measure computed over the top set of results. This helps identify documents that are more “central” or representative among all query matches.
4. Document Recommendations
    - Suggests additional books that share thematic or vocabulary overlap with the user’s query. This is achieved using a lightweight Jaccard similarity calculation between the query’s token set and each document’s token set. The recommendation system provides an easy way to explore the corpus and find related material without performing new searches.

All heavy computations that would slow down interactive use, such as tokenization, vectorization, indexing, vocabulary extraction, and TF-IDF generation, are

performed offline at backend startup. Queries operate only on in-memory data structures, ensuring the response time stays consistently low even when processing complex searches or working with large result sets. This design also allows the backend to handle multiple users simultaneously without significant degradation in performance.

This report presents the algorithms, data structures, and design decisions used in the system. The focus is on how textual data is indexed, how queries are processed, how ranking and recommendation are computed, and how these components work together to form a coherent retrieval pipeline. Performance measurements and practical evaluations highlight the strengths and trade-offs of each approach. The goal is to provide a complete and technically grounded description of a search engine built for large textual collections, demonstrating how careful preprocessing and efficient data modeling directly influence the responsiveness and quality of search results.

Implementation

Our problem is that at a high level, the system's purpose is to take a user's free-form query and determine which documents in the corpus are most relevant, how they should be ranked, and which additional texts might be related. Achieving this requires several capabilities: efficiently representing long documents, retrieving matches quickly without rescanning the raw text files, ordering those matches in a meaningful way, supporting pattern-based queries through regex, and identifying documents that are similar in theme or vocabulary. The core problem therefore becomes creating a retrieval pipeline that balances speed, accuracy, and scalability while remaining simple enough to serve results in real time through a web API.

A search engine built on long, irregular, and vocabulary-heavy texts requires data structures that allow fast access, efficient comparisons, and reliable ranking. The system must avoid scanning raw text, must keep all essential structures in memory, and must separate presentation data from computational data. The following data structures form the foundation of the search, ranking, and recommendation pipeline. Each one was chosen because it works well with large collections of documents while staying lightweight enough for real-time querying.

Key operations

The desired operations give way to our chosen data structures. In this case, we must be able to:

1. For looking, get a list of documents mentioning an indexed term, and be able to order these documents by how often the term is referenced.
2. For closeness centrality ranking, encode each document as a vector of equal size.

To solve these problems at the same time, we use a TF-IDF matrix, where rows represent documents, columns represent terms. This way, each row is a directly comparable document vector, and a search can be done by comparing TF-IDF values across the document set for a limited number of columns dictated by the query.

TF-IDF matrix

The core data structure used by the search engine is a TF-IDF matrix created with scikit-learn's TfidfVectorizer. Each document becomes a sparse numerical vector where each position corresponds to a vocabulary token and each value represents its TF-IDF score. Sparse representation is crucial here: long documents share very little vocabulary, so storing the vectors densely would waste memory and slow down computations. Prior to building the matrix, simple noise reduction is applied by removing very short tokens (typically ≤2 characters) and ignoring common stop words that carry little semantic weight. These steps reduce dimensionality without harming retrieval accuracy. The TF-IDF matrix is built once during backend startup. If the corpus has been modified by adding or removing files, the system regenerates the matrix to keep the index consistent. With the vocabulary capped at 100,000 unique features, the matrix remains compact enough to fit fully into memory. For a corpus of around 1,947 documents, this provides fast search operations without exceeding reasonable compute or memory limits.

TF-IDF offers a reliable balance between simplicity and relevance. It downweights very common terms while highlighting words that distinguish documents from one another. It integrates smoothly with cosine similarity, which is fast to compute even in high-dimensional sparse spaces. Although TF-IDF does not capture word order or semantic meaning, and will not detect synonyms or related concepts, it remains an effective and efficient baseline for keyword-based retrieval. The main limitation is that TF-IDF is purely lexical. It does not understand semantic relationships such as *doctor* ≈ *physician*, nor does it capture longer contextual information. Still, its performance, speed, and interpretability make it a strong fit for this project.

Document vector

In addition to the vector representation, each document is associated with a token set, generated using the same analyzer as the TF-IDF vectorizer. For a document d:

$$\text{tokens}(d)=\{w1,w2,\ldots,wn\}$$

Token sets serve two main functions:

- Regex Term Expansion:
  - Regex queries are applied to the vocabulary itself, not the document text. This prevents the system from scanning full files, which would be extremely slow. Matching regex patterns directly on the token list keeps the operation efficient even with a large vocabulary.
- Jaccard-Based Recommendations:
  - For recommendations, we compare the query's token set to each document's token set using simple set operations: Insertion, Union and Overlap ratio.

Python's optimized set implementation allows these operations to run very quickly.

Set intersections and unions are extremely fast and scale gracefully with token count. They provide a simple structure for determining overlap between documents, which is effective for identifying thematically related texts. Token sets also allow regex matching without ever touching the raw documents. Token sets ignore word frequency. For recommendations, this is acceptable since the goal is to detect topical similarity rather than precise ranking. However, for ranking itself they are insufficient, which is why TF-IDF is still required for the primary search logic.

In-memory operations

Rather than relying on a database or an external indexing service, document metadata is stored in a simple JSON file. Each entry includes:

- Document ID
- Title and author
- File path to the raw text
- Cover image URL

This design keeps the retrieval pipeline efficient by separating display-related information from search-related information. Search algorithms operate exclusively on TF-IDF vectors and token sets, while metadata is retrieved only when finalizing the response sent to the frontend. JSON keeps the system lightweight. The metadata is read once at startup and stored in memory alongside the index structures. A database would not improve performance in this context because the amount of metadata per document is small and infrequently modified.

The corpus and all index-related data are separated into two directories:

- Raw Text: All downloaded .txt files from Project Gutenberg, each named using the Gutenberg ID, making them easy to locate and update.
- Index Directory: Stores the TF-IDF matrix, token set cache, inverted index, and metadata file.

This separation allows the system to rebuild these indexing components solely using the JSON metadata file, reducing repeated I/O on raw text files and improving reproducibility and maintainability.

Keyword matching and ranking

The search engine relies on a set of algorithms designed to identify relevant documents quickly and rank them in a meaningful way either as search results or as recommendation results of similar documents. These algorithms operate on the data structures described previously together form the core of the retrieval pipeline.

Keyword search is the primary retrieval method. When a user enters a free-form query, the backend tokenizes it using a CountVectorizer analyzer. Only query terms that exist in the TF-IDF vocabulary are retained, preventing unnecessary operations on unknown tokens. Once query terms are extracted, the system uses the TF-IDF matrix to locate the top documents in which these terms have the highest weights. This is implemented using the pandas .nlargest() function, which retrieves the rows (documents) with the highest TF-IDF scores for the selected columns (query terms). The system then filters out documents with zero total contribution to ensure that only relevant hits remain. This approach scales well because the core operation, finding the largest values along specific sparse vector dimensions is efficient even for large matrices. Since the matrix is already in memory, keyword search avoids disk I/O entirely. This search method has many advantages such as:

- Very fast even for multi-term queries

- Returns results ordered by how strongly documents match the query
- Benefits directly from TF-IDF's downweighting of common words

Despite these strengths there are a few drawbacks such as:

- Queries consisting of rare inflected words or synonyms may not match the vocabulary if the exact token is not present
- Relevance is purely lexical, not semantic (example doctor ≈ physician)

Given these limitations, TF-IDF keyword search provides a solid and efficient baseline for retrieving meaningful results across a large corpus quickly when cached already.

Regex matching

Pattern-based search is handled by applying the user's regular expression only to the vocabulary, not the raw documents. This design ensures that regex operations are quick: instead of scanning millions of characters, the system simply checks which tokens in the TF-IDF index match the pattern. Once matching tokens are identified, the algorithm treats them the same way as keyword search: using .nlargest() to find documents with high TF-IDF scores for those terms. For example, the regex: "king.*" may match tokens like king, kings, kingdom, and kingly, each contributing to the retrieval process. This process:

- Is extremely fast compared to full-text regex search
- Enables pattern-based exploration of the corpus
- Works well with morphological families of words

Overall, applying regex to the vocabulary is a practical compromise between flexibility and performance with a few drawbacks such as:

- Regex cannot capture semantic patterns, only literal ones
- Overly broad regex patterns may yield large token lists, increasing noise
- Does not consider word positions or larger text structure

Closeness centrality ranking

The default results ranking simply uses TF-IDF total to order documents from the matrix, which is the direct output of the .nlargest() method. We additionally implement different

rankings, namely closeness centrality, with the framework to implement more rankings if desired.

Closeness centrality ranking works by ordering the documents based on the minimum total “distance” between one document and each other document in the result set. This means the first step is to build the result set itself, which works the same way as with occurrence ranking: simply taking all documents matching at least one query term (TF-IDF total > 0). From here, we define the distance from one document to another to be the cosine similarity between the two document vectors. This means to compute closeness centrality, we must find the similarities between each pair of two documents in the result set. This is the other key area where .nlargest() comes into play: we limit all queries to the top 50 results because squaring the size of the result set makes closeness centrality ranking very slow when there are many matches. From here, a new vector is created for each document containing the similarities between it and each other document in the set, and the final result order is dictated by the sum of each vector in this set, in descending order.

Recommendations

The recommendation system provides users with additional documents that share topical similarity with the query they entered, enabling exploratory navigation of the corpus beyond the direct search results. Unlike the main ranking pipeline, which relies on TF-IDF vectors and cosine similarity, the recommendation system uses an interpretable set-based similarity measure. This design keeps recommendations fast to compute, scalable to larger corpora, and semantically meaningful when queries consist of multiple related terms.

Each document in the corpus is preprocessed at backend startup to extract a unique token set:

$$\text{tokens}(d)=\{w_1, w_2, \ldots, w_n\}$$

Tokenization is performed using the same analyzer as the TF-IDF vectorizer, ensuring consistent preprocessing (lowercasing, punctuation handling, stop-word removal). The resulting token set acts as a compact thematic fingerprint of each book, capturing the main vocabulary without preserving ordering or frequency. These token sets are computed once, cached in memory, and reused for all recommendation queries. This eliminates repeated disk I/O and ensures that recommendation latency remains negligible.

To compare the user’s query with each document, the system uses Jaccard similarity, a classical measure of set overlap:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Here,

- A = token set of the user’s query
- B = token set of a document

The metric ranges from 0 to 1, where 1 indicates identical token sets. In practice, even small overlaps can meaningfully capture shared topics (e.g., “kingship”, “royalty”, “monarch”). The use of Jaccard similarity is appropriate for this task because: It captures vocabulary overlap, which strongly correlates with topical similarity. It avoids TF-IDF’s bias toward long documents. It is extremely efficient due to Python’s optimized set operations.

The recommendation process consists of four stages:

- Tokenize the query: The input query is processed with the same CountVectorizer analyzer used during indexing, producing a normalized set of query tokens.
- Compute Jaccard similarity: For each document, compute the size of the intersection and union with the query’s token set. Documents with zero overlap are discarded immediately.
- Sort by similarity: All non-zero-similarity documents are sorted in descending order by their Jaccard score.
- Return the top results: The top eight documents are returned as recommendations. This cutoff balances quality and variety while keeping the response lightweight. This pipeline completes in a few milliseconds, owing to the preloaded token sets and inexpensive set operations.

This pipeline completes in a few milliseconds, owing to the preloaded token sets and inexpensive set operations.

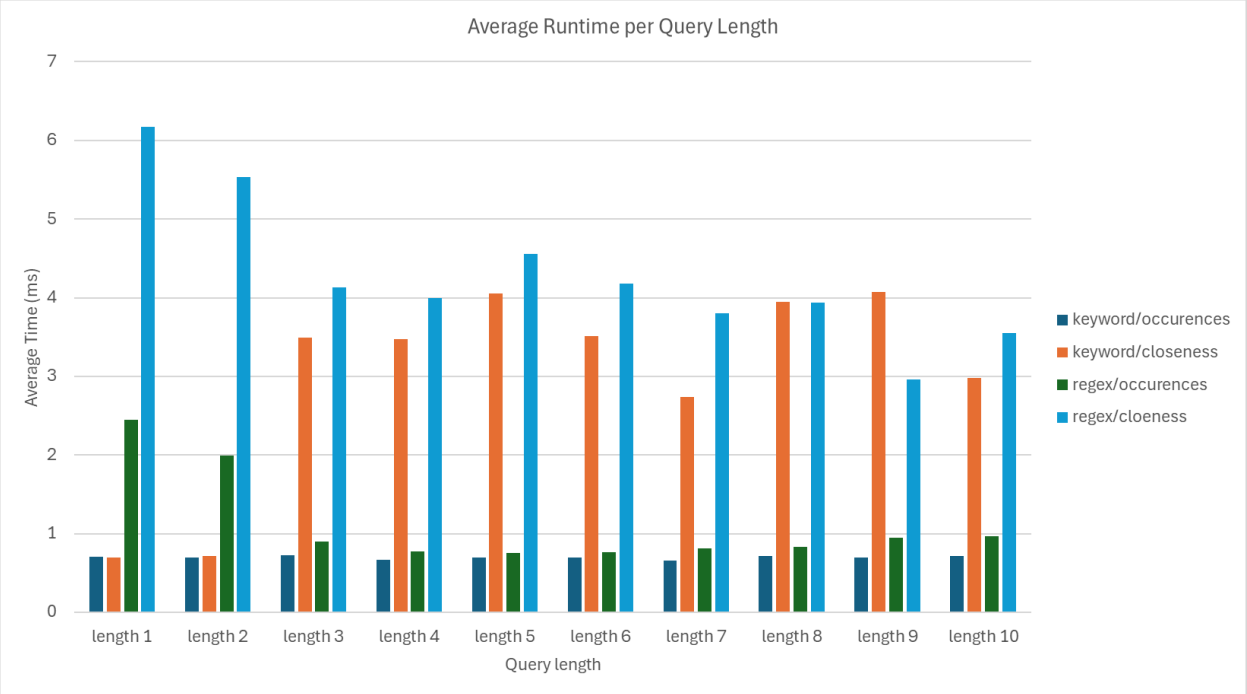
The recommendation system performs well across diverse types of queries, especially multi-term and thematic ones. Because it does not rely on TF-IDF magnitudes, it avoids overweighting documents that happen to repeat the same word frequently. Instead, it rewards broad vocabulary overlap, which aligns better with the notion of “similar topics.”

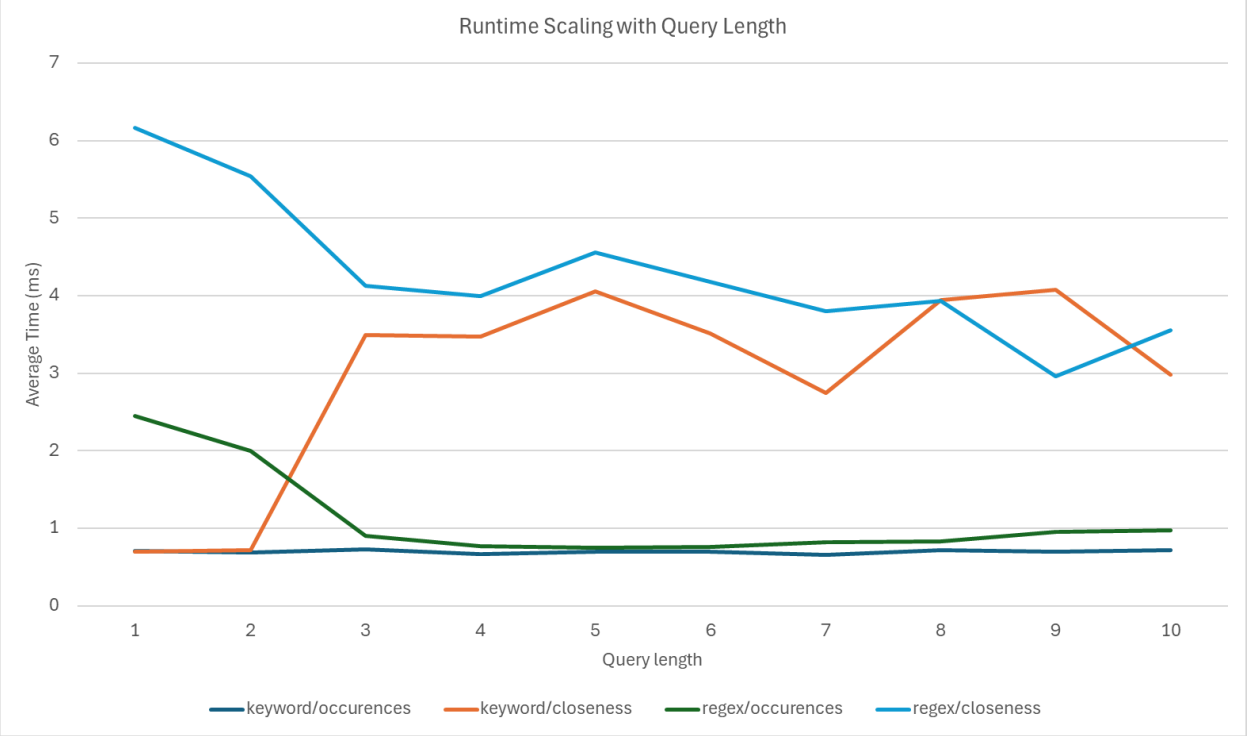
However, several limitations remain:

- Jaccard similarity still does not account for semantic relationships.
- Token sets ignore frequency, which sometimes discards useful information about emphasis.
- Long queries with many rare words may result in low overlap and sparse recommendations.

Despite these limitations, the Jaccard-based approach provides an effective and computationally lightweight method for discovering related books in a large corpus. It complements the main keyword and regex search pipelines by offering users a secondary navigational path based solely on vocabulary similarity.

Performance analysis and testing





To ensure that performance measurements reflected realistic usage conditions, all tests were executed on the same machine under consistent system load. Before each test sequence, the backend completed its startup process—loading the TF-IDF matrix, token sets, and

metadata into memory—so that all measurements captured only the cost of the query pipeline itself, not initialization overhead.

To evaluate the behavior of each search pathway, we tested four modes:

- Keyword search ranked by occurrences
- Keyword search ranked by closeness centrality
- Regex search ranked by occurrences
- Regex search ranked by closeness centrality

For each query length $\ell \in \{1,\ldots,10\}$, 6 queries were generated and executed under all four modes. As each query was executed, the average execution time (in milliseconds) was recorded. This produced a dataset of 240 averaged measurements, from which the charts above were constructed. All experiments used the same corpus of 1,947 Gutenberg documents and the same vocabulary extracted by TfidfVectorizer. As well, this data is collected without including the api RTT time only showing actually algorithm cost

14 charts were constructed from the measurements, following that we selected the above 2 charts to show the performance analysis.

- Average Runtime per Query Length (Bar Chart) (top chart): This grouped bar chart compares absolute execution times across the four search modes for each query length. It reveals how the methods differ in computational cost and highlights the effect of adding closeness centrality on top of a baseline search.
- Runtime Scaling with Query Length (Line Chart) (bottom chart): This line chart captures how each method behaves as query length increases from 1 to 10 tokens. It visualizes trend behavior, stability, variance, and algorithmic scaling more clearly than absolute bar comparisons.

Across both charts, several consistent patterns emerge:

1. Keyword search is consistently fast, with mild variance. Keyword-occurrence search stays between ~3–5 ms across all lengths. The TF-IDF matrix gives O(1) access to each required column, and aggregating contributions across a fixed number of rows is extremely cheap. The small rise and fall across lengths comes mostly from slight differences in how many documents match each query.
2. Closeness centrality is the main cost driver. Both keyword and regex searches become significantly slower when closeness is applied. This matches the theoretical cost: $O(k^2)$ where k = 50 top documents. Each closeness computation requires forming a 50×50 cosine similarity matrix. Even with sparse TF-IDF vectors, this dominates runtime and accounts for the ~3–6 ms range seen in the bar chart.
3. Query length has little effect on regex cost and only modest effect on keyword cost. The line chart clearly shows that: Regex curves are nearly flat. Keyword curves decline slightly as queries become more specific (usually by using more descriptive words compared to short words that are found often). Closeness curves track keyword behavior but with higher absolute cost. This means the

system scales well with longer user queries and preserves interactive responsiveness across realistic usage patterns.

4. Overall system responsiveness meets real-time web requirements. All four methods complete well under 10 ms—even closeness centrality at its slowest. This comfortably fits within the typical 16–30 ms budget for a smooth front-end interaction loop and ensures that even complex queries return instantaneously from the user's perspective.

Conclusion

For this project, a web application implementing search for a subset of documents from Project Gutenberg was implemented. Multiple search types were implemented, as well as multiple ranking methods, which are separately composable to customize the search experience. Additionally, our default corpus of just under 2000 documents can be easily expanded as desired, with auto-indexing providing a smooth application experience.

While we were able to implement all of our target features, there are still some things left to be desired. Most notably, the limitation of at most 50 search results is not configurable due to closeness centrality ranking. It should be possible to use tricks to more efficiently compute this kind of ranking on larger datasets, as well as allowing multiple pages of results for traditional occurrence ranking. Additionally, the lack of semantic meaning in search makes this not the most dynamic for real-world use cases, although it closely implements what we have seen in class. We believe that at least our matrix representation would adapt relatively smoothly to other document vector representations which encode meaning using ML techniques, leaving this possibility open for the future. Overall, this project was an excellent learning and problem-solving opportunity, and we are happy with the concrete result we were able to obtain.