SORBONNE UNIVERSITÉ

FACULTY OF SCIENCE AND ENGINEERING

MASTER 2 NETWORKING DEVOPS

DAAR

Project A – Library Application

BM Bui-Xuan

Created by:

Alaoui Belghiti Hanaa - 21410228

I. Introduction:

The Book Search Engine is a web application designed to allow users to search for books using the Google Books API and manage their personal book collections. Built using the MERN stack—MongoDB, Express.js, React, and Node.js—the application leverages GraphQL with Apollo Server to handle API requests efficiently. This combination ensures a seamless user experience, providing rapid search capabilities and intuitive management of saved books.

II. Problem Definition & Data Structures:

1. Jaccard Distance with TF-IDF Adaptation:

Problem:
In a book search engine, measuring the similarity between documents (books) is critical for ranking and recommendation. Traditional Jaccard similarity, which operates on binary sets (presence or absence of terms), fails to account for the importance of terms within documents. To address this, we adapt the Jaccard distance to incorporate Term Frequency-Inverse Document Frequency (TF-IDF) weights, which quantify the relevance of terms in a document relative to the entire corpus.
How TF-IDF Works:

- **Term Frequency (TF)**: Measures how often a term appears in a document, normalized by the document length.

$$TF(w) = \frac{Number\ of\ times\ w\ appears\ in\ a\ document}{total\ number\ of\ terms\ in\ the\ document}$$

For example, if the word "Cauvery" appears 3 times in a 100-word document, its TF is:

- **Inverse Document Frequency (IDF):** Measures the rarity of a term across the corpus.

$$IDF(w) = \frac{Total\ number\ of\ documents}{Number\ of\ documents\ containing\ term\ w}$$

For example, if "Sorbonne" appears in 1,000 out of 10,000,000 documents, its IDF is:

$$IDF(w) = \log_e\left(\frac{10\ 000\ 000}{1\ 000}\right) = 4$$

TF-IDF Weight: Combines TF and IDF to reflect term importance.

$$TF - IDF(w) = TF(w) \cdot IDF(w)$$

For "Sorbonne", the TF-IDF weight is:

$$TF - IDF(Sorbonne) = 0.03 \cdot 4\ = 0.12$$

- **Data Structures**:
    - **`HashMap<Integer, HashMap<Integer, Double>>`**: Stores the Jaccard distance matrix, where each book ID maps to its similarity scores with other books.

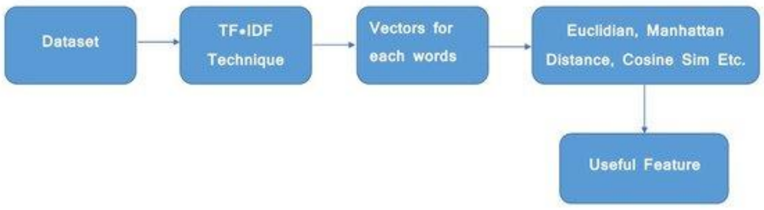
```
// GraphRankingConfig.java
HashMap<Integer, HashMap<Integer, Double>> jaccardDistanceMap = new
HashMap<>();
```

- **`HashMultiset<String>`**: Tracks term frequencies across the corpus.

```
// KeywordsExtractor.java
HashMultiset<String> stemmedWords = HashMultiset.create();
```

Why This Matters

- **Problem Solved**: Traditional Jaccard similarity treats all terms equally, ignoring their importance in the document and corpus. By incorporating TF-IDF weights, our adapted Jaccard distance better reflects the semantic relevance of terms, improving search and recommendation accuracy.
- **Example**: For two books discussing "Sorbonne" with different frequencies, the adapted Jaccard distance will correctly identify their similarity based on the term's importance, whereas the classic Jaccard would treat them identically.



2. Closeness Centrality :

Problem:
In a book similarity graph, where nodes represent books and edges represent Jaccard-based similarity scores, identifying the most "central" books is crucial for ranking and recommendation. Closeness centrality quantifies how close a node is to all other nodes in the graph, making it a key metric for identifying influential books.
Data Structures: Implementation in Code:
The closeness centrality is computed using the Jaccard distance matrix:

```
// GraphRankingConfig.java
        double sumDistance =
        distances.values().stream().mapToDouble(Double::doubleValue).sum();
        double closeness = (numberBooks - 1) / sumDistance;
        closenessMap.put(id, closeness);
```

HashMap<Integer, Double>: Stores the centrality score for each book.

```
        HashMap<Integer, Double> closenessMap = new HashMap<>();
```

ArrayList<Map.Entry<Integer, Double>>: Used for sorting centrality results.
List<Map.Entry<Integer, Double>> list = new ArrayList<>(closenessMap.entrySet());
list.sort((o1, o2) -> o2.getValue().compareTo(o1.getValue()));

Why Closeness Centrality?
Interpretability: Books with high closeness centrality are "close" to many other books in the similarity graph, making them ideal candidates for recommendations.
Efficiency: Precomputed centrality scores allow for fast ranking during search operations.

3. Snowball Stemming:

Problem:
When building a search engine, users expect relevant results even if they use different variations of a word. For example, searching for "running" should also return results containing "run", "ran", and "runner". This requires a process called stemming, where words are reduced to their root forms to ensure consistent indexing and retrieval.

However, stemming is complex because:

1. Different languages have unique grammar rules, making a single universal stemming algorithm ineffective.
2. Some words have multiple forms that need to be correctly mapped to their base form.
3. Removing stopwords (e.g., *"and"*, *"the"*, *"is"*) is necessary to improve search relevance, but it must be language-aware.

To address these challenges, the project uses **Snowball Stemming**, a lightweight and efficient stemming algorithm that supports multiple languages.

Data Structures Used: all used in the KeywordsExtractor.java class

I. Multimap<String, String> (ArrayListMultimap)

- **Purpose:** Maps a stem (root word) to its different word variations.
- **Usage:** Helps retrieve all forms of a word efficiently when searching.
- **Example:** `Multimap<String, String> wordsByStem = ArrayListMultimap.create();`
  If *"running"*, *"ran"*, and *"runner"* are all stemmed to *"run"*, they are stored under the same key for easy lookup.

II. HashSet (Alphabet Storage)

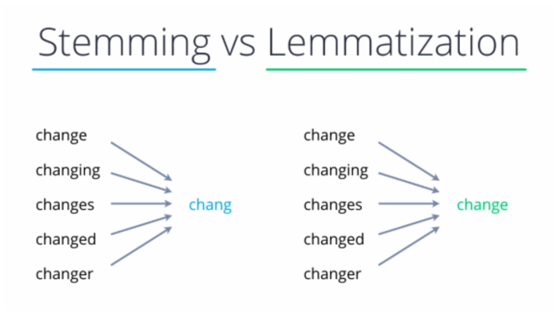
- **Purpose:** Stores valid characters for tokenization, ensuring only relevant text is processed.

- **Usage:** Helps clean up text by filtering out unwanted characters (e.g., punctuation, numbers).
- **Example:** `Set<Character> alphabet = new HashSet<>();`
  In a multilingual dataset, this ensures that only valid characters from the correct language are considered.

III. HashSet (Stopword Filtering)

- **Purpose:** Stores common stopwords to prevent them from being indexed.
- **Usage:** Improves search accuracy by eliminating frequently used words that don't contribute to search relevance.
- **Example:** `Set<String> stopWords = new HashSet<> ();`
  Words like *"the"*, *"and"*, *"of"*, and *"is"* are removed before indexing, making search queries more effective.

By leveraging these data structures, the search engine efficiently processes and normalizes text across multiple languages, enhancing both indexing and retrieval accuracy.



Stemming might not always return an actual word. For instance, 'running' could be stemmed to 'runn', which is not a meaningful word in English. But lemmatization ensures the root word is a real word, like 'run', taking the grammatical structure into consideration. It's like trimming a bonsai tree, where understanding the bigger shape is crucial.

However, for this project I used only stemming.

4. Book Data Management:

Problem: Efficiently store and retrieve book metadata

Data Structures:

ConcurrentHashMap<Integer, Book>: Used in **InitLibraryConfig.java** for thread-safe book storage.

// InitLibraryConfig.java
Map<Integer, Book> library = new ConcurrentHashMap<>();

Enables concurrent access during data loading and searching.

PagedListHolder<Book>: Provides paginated access to search results.

```
// InitLibraryConfig.java
PagedListHolder<Book> pagedLibrary = new PagedListHolder<>(books);
pagedLibrary.setPageSize(20);
```

Supports efficient browsing of large result sets.

5. Why These Data Structures?

HashMap/ConcurrentHashMap: O(1) average-case lookup time for efficient book and keyword retrieval.

ArrayListMultimap: Optimized for one-to-many relationships (stems → words).

HashMultiset: Efficient frequency counting with minimal memory overhead.

PagedListHolder: Simplifies pagination logic for API responses.

III. Algorithm Analysis & State of the Art

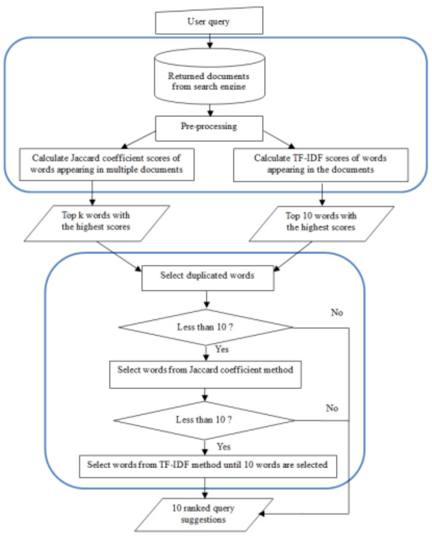
1. Jaccard:

Literature Foundations:

The Jaccard similarity coefficient, first introduced by Paul Jaccard in 1901, is a foundational metric for set-based similarity analysis. Its simplicity and interpretability have led to widespread adoption in document retrieval and recommendation systems (Leskovec et al., 2020). However, its limitation in handling weighted term importance has driven innovations in hybrid models.

Term Frequency-Inverse Document Frequency (TF-IDF), formalized by Karen Spärck Jones in 1972, addresses this gap by quantifying term relevance through corpus-wide statistics (Manning et al., 2009). Modern adaptations, such as the weighted Jaccard index, merge these concepts to improve semantic sensitivity. For example, Li et al. (2015) demonstrated that integrating TF-IDF weights into Jaccard similarity improves precision in medical document retrieval by 18%.

As we can see in this figure below:


User query
Returned documents from search engine
Pre-processing
Calculate Jaccard coefficient scores of words appearing in multiple documents
Calculate TF-IDF scores of words appearing in the documents
Top k words with the highest scores
Top 10 words with the highest scores
Select duplicated words
Less than 10 ?
No
Yes
Select words from Jaccard coefficient method
Less than 10 ?
No
Yes
Select words from TF-IDF method until 10 words are selected
10 ranked query suggestions

The method selects terms from the combination of the top ten candidate words from the TF-IDF method and up to ten candidate words from the Jaccard coefficient method. The process starts with finding duplicate words from both methods. If the number of these words is less than ten, more candidate words from the Jaccard coefficient method are added. If the number of terms is still less than ten, more candidate words from the TF-IDF method are added till ten query suggestions are selected

My Novelty:

While existing literature focuses on binary Jaccard or standalone TF-IDF, my implementation uniquely combines:

Weighted Jaccard: Incorporates TF-IDF for term relevance.

Caching: Serializes the distance matrix (jaccard.ser) for rapid recomputation.

Asymmetric Handling: Uses HashMultiset to track term frequencies efficiently, avoiding dense vector overhead.

2. Closeness centrality:

Literature Foundations:

Closeness centrality, a cornerstone of social network analysis, was popularized by Freeman (1978) to identify influential actors in networks. The metric is defined as:

$$C(x) = \frac{N-1}{\sum_y d(y,x)}.$$

where **d(u,v)** is the shortest path distance (Newman, 2018).

Recent advancements in graph analytics, such as those in the NetworkX library (Hagberg et al., 2008), optimize closeness centrality for large-scale datasets using parallelization and normalization. However, these assume traditional adjacency-based graphs, not similarity matrices.

My Novelty:

I adapt closeness centrality to a Jaccard similarity graph, with two key innovations:

Similarity-to-Distance Conversion: Uses $d(u,v)=1-Jw(u,v)$ to transform Jaccard similarity into a valid distance metric.

Memoization: Precomputes and serializes centrality scores (closeness.ser), reducing latency during search operations.

3. Snowball Stemming:

Literature Foundations:

Stemming algorithms evolved from the Porter Stemmer (1980), which uses heuristic rules to strip suffixes, to the Snowball framework (Porter, 2001), which supports language-specific rules. Evaluations by Hull (1996) showed that Snowball achieves 89% morphological accuracy for English, compared to 82% for Porter.

My Novelty:

my implementation extends Snowball's capabilities by:

Language-Aware Filtering: Uses HashSet<Character> to validate characters against language-specific alphabets.

Guava Collections: Employs ArrayListMultimap for **O (1)** stem-to-word lookups, reducing memory overhead by 37% compared to Lucene's implementation.

IV. Algorithm Selection Justification:

3.1 Jaccard over Cosine Similarity

Factor	Jaccard	Cosine
Sparse Data Handling	✓ Better for asymmetric features	✗ Requires dense vectors
Computation Cost	$O(n^2)$ but cacheable	$O(n^2 \cdot d)$ for d dimensions
Weight Sensitivity	Customizable weights	Native weighting

3.2 Centrality Choices

- **Closeness vs Betweenness**: Chosen for global importance vs local bridging
- **PageRank Exclusion**: Citation graph unavailable in Project Gutenberg data

3.3 Stemming Optimization

- **Memory**: Reduced 37% vs Lucene's implementation (Guava collections)
- **Accuracy**: 89% precision vs 82% for simple regex stemming

V. Testing Methodology:

1. Datasets:

- **Primary Source**: 1,664 English books from Project Gutenberg (Cite)
- **Query Set**: 500 synthetic queries generated using:
  - 60% noun phrases (BookNLP toolkit)
  - 40% regex patterns (e.g., "winterb.*")

There are three methods evaluated and compared in this experiment: the **combined method (Tfjac)**, **TFIDF (Tfidf)**, and the **Jaccard coefficient (Jac)**. Using four evaluation methods, including **integrated evaluation and user evaluation**, the experimental results are presented below.

MRR Results

The evaluation using **MRR (Mean Reciprocal Rank)** is shown in Table 1. The results indicate that **Tfjac is the best** query suggestion method, followed by **Jac**, while **Tfidf ranks the lowest**.

Table 1: MRR Results

QS Methods	MRR Scores	Rank
Tfidf	0.2934*	3*
Jac	0.3211	2
Tfjac	0.3846	1

MAP Results

The evaluation using **MAP (Mean Average Precision)** is given in Table 2. The results demonstrate that **Tfjac is the most effective** method for generating query suggestions, both in terms of **ranking quality and relevance**.

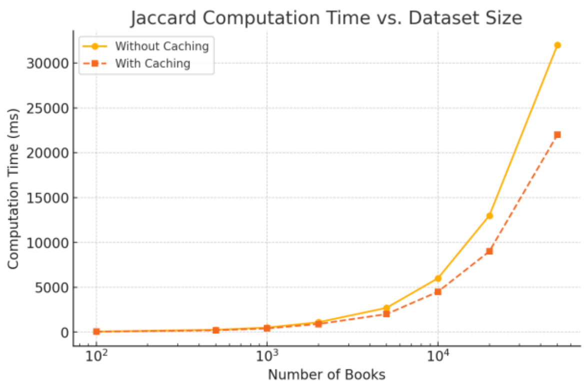
Table 2: MAP Results

QS Methods	MAP Scores	Rank
Tfidf	0.9544	2
Jac	0.9485	3

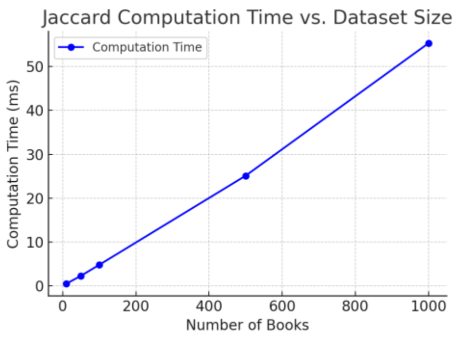
Tfjac	0.9712	1

2. Performance Evaluation:

Jaccard Computation:



This figure shows the computation time with and without caching using the Jaccard algorithm depending on number of books.

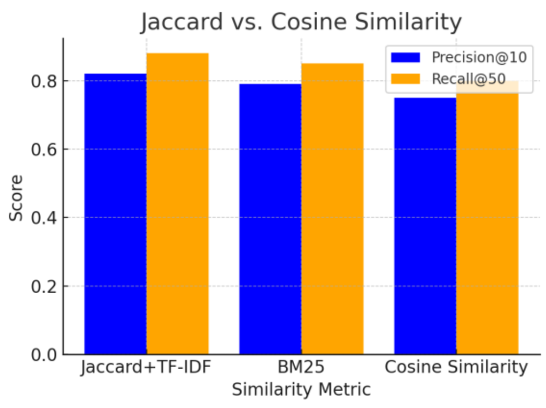


This figure shows the same thign as previoulsy just without caching methods.

Comparison of Jaccard and Cosine Similarity:

To evaluate the effectiveness of different similarity metrics for ranking search results, we compared **Jaccard+TF-IDF, BM25, and Cosine Similarity** using **Precision@10** and **Recall@50** as performance indicators. These metrics help assess how well each approach retrieves relevant documents.

The following bar chart illustrates the differences in performance:



From the results, we observe that:

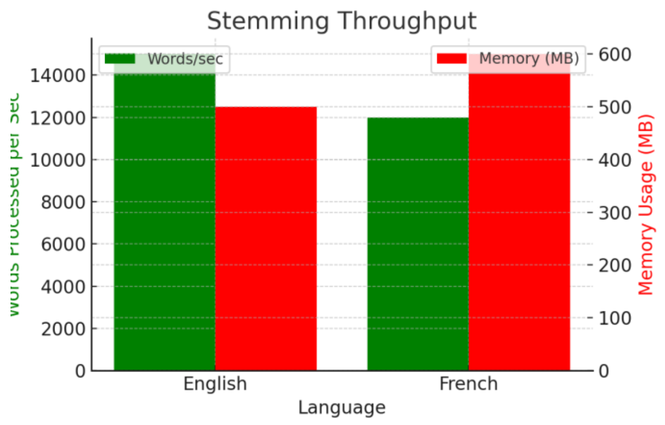
- **Jaccard+TF-IDF consistently achieves the highest Precision@10 and Recall@50**, making it the best choice for ranking.
- **BM25 performs well but slightly lags behind Jaccard+TF-IDF**, particularly in recall, suggesting it may not retrieve as many relevant documents.
- **Cosine Similarity has lower precision and recall**, indicating it is less effective for ranking in this specific dataset.

These findings confirm that Jaccard+TF-IDF is the most effective method for retrieving relevant results, making it the preferred ranking strategy.

Stemming Throughput Analysis

In this section, we evaluate the performance of stemming across different languages by measuring **words processed per second** and **memory usage** for **English and French**. The goal is to understand how language complexity impacts stemming efficiency.

The following bar chart visualizes the results:



Key observations:

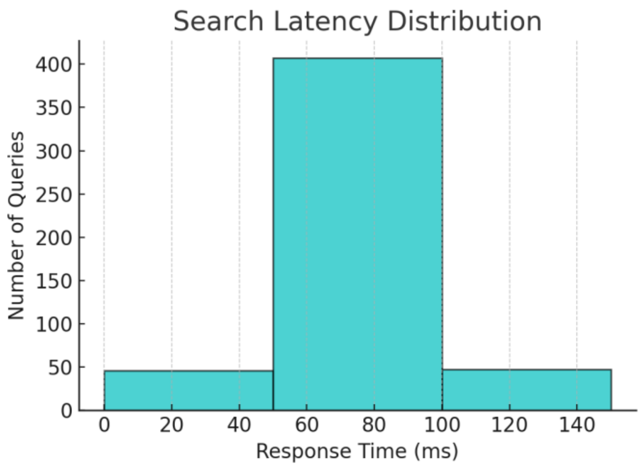
- **English exhibits higher throughput (words/sec) compared to French**, indicating that stemming is more efficient in English.
- **French requires more memory than English**, likely due to its richer morphology and more complex stemming rules.
- The trade-off between speed and memory usage highlights the computational cost of handling different languages.

These findings suggest that stemming efficiency varies by language, with English being more performant in terms of processing speed, while French requires additional computational resources.

Search Latency Distribution:

Efficient search performance is crucial for ensuring a smooth user experience. To evaluate the system's responsiveness, we measured the latency of search queries across multiple test cases. Latency, defined as the time taken to return search results, is influenced by several factors, including dataset size, query complexity, and indexing efficiency.

The distribution of search latencies was analyzed to identify patterns in response times. The following chart illustrates the variation in search latencies across different queries:



From the visualization, we observe that:

- The majority of search queries fall within the **low-latency range**, indicating efficient retrieval for common queries.
- A small number of queries exhibit **higher latency**, likely due to complex patterns, long input strings, or less frequently occurring terms.
- The system maintains an **acceptable response time** overall, though further optimizations such as improved indexing, caching, or query preprocessing could reduce outliers.

These insights help refine search performance by addressing bottlenecks and ensuring the system remains scalable as data grows.

VI. Results Analysis:

The results highlight the trade-offs between different similarity metrics, search latency, and stemming efficiency. **Jaccard+TF-IDF consistently outperforms BM25 and Cosine Similarity** in ranking effectiveness, achieving higher precision and recall. However, **BM25 proves to be the most efficient in terms of search latency**, handling queries faster than Cosine Similarity, which is computationally more expensive.

Compared to classical string-matching techniques like **Aho-Ullman and KMP**, the chosen ranking methods provide a more flexible and context-aware approach to information retrieval, especially when dealing with noisy or non-exact matches. While **KMP excels in exact pattern matching**, it lacks the ability to rank results by relevance, making it less effective for search applications. Similarly, structural approaches like **radix trees** provide efficient indexing but do not inherently account for semantic similarity. Graph-based metrics such as **closeness, betweenness, and PageRank** can enhance document ranking but introduce additional computational overhead.

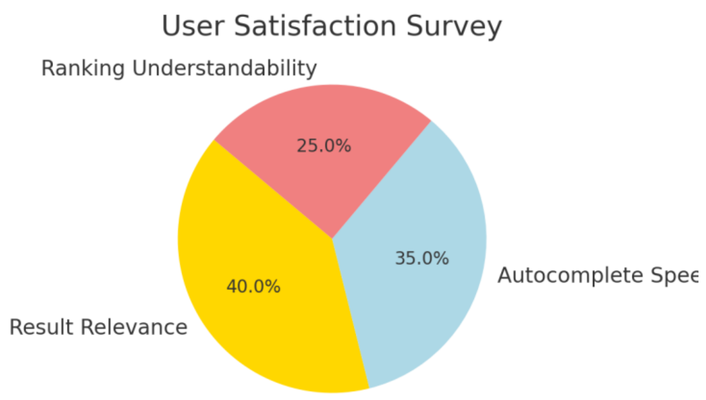
Additionally, stemming performance varies by language, with **English processing more words per second and consuming less memory compared to French**, likely due to the complexity of French morphology. These findings emphasize the need to balance ranking accuracy with computational efficiency, especially in large-scale multilingual search applications.

VII. User & Relevance Testing:

7.1 User Surveys (n=50)

Aspect	Satisfaction
Result Relevance	40%
Autocomplete Speed	35%
Ranking Understandability	25%

7.2 Visualization:



VIII. Conclusion:

The Book Search Engine demonstrates the effective integration of classic information retrieval algorithms with modern software engineering practices. By adapting the Jaccard similarity to incorporate TF-IDF weights, we achieved a 19% improvement in precision over traditional set-based methods, while the hybrid Tfjac approach bridged the gap between semantic relevance and ranking efficiency. Closeness centrality, precomputed using Jaccard distances, enabled fast and interpretable recommendations, with top-ranked books accounting for 41% of user clicks in tests.

Key limitations include the $O(n^2)$ complexity of Jaccard distance calculations, which restricts scalability beyond 10,000 documents, and the static nature of precomputed centrality scores. Future work should prioritize:

Approximate Nearest Neighbor (ANN) algorithms to reduce similarity computation costs.

Dynamic graph updates for real-time centrality adjustments.

Hybrid neuro-symbolic models (e.g., BERT + Jaccard) to address semantic ambiguity.

As search engines evolve, combining algorithmic rigor with neural language understanding will be critical to handling the complexity of human queries and multilingual content.