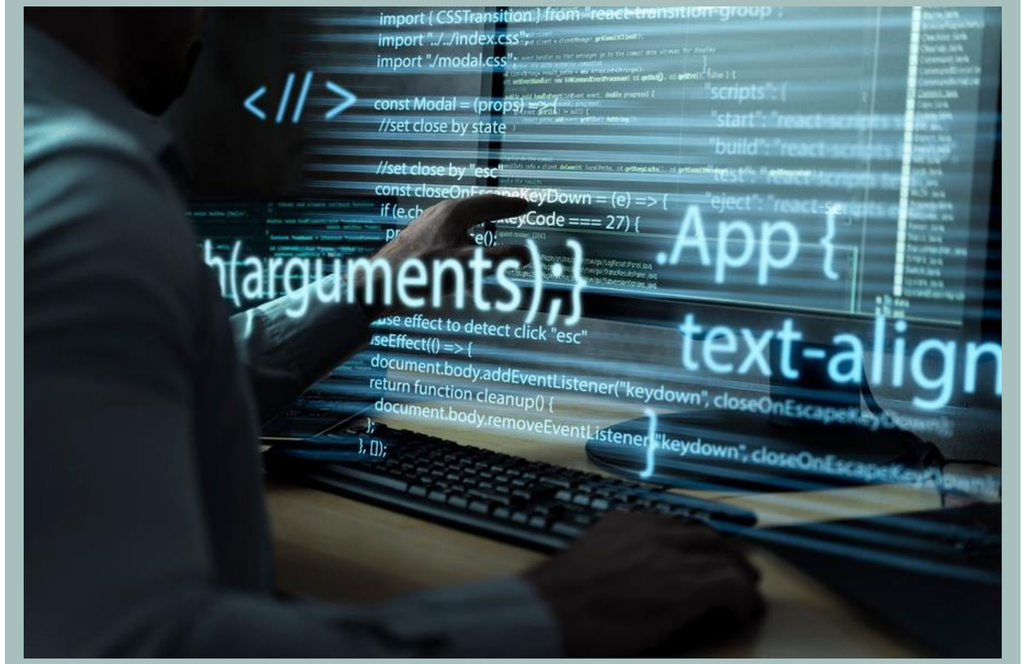
Search Engine

Done by:

Radoslawa Zukowska
Anna Sajdokova
Tymoteusz Oczowiński
José Luis Perdomo de Vega
Eduardo López Fortes



Index

Introduction

- Context: Efficient search in large data volumes is crucial in today's information-driven world, and search engines are fundamental tools for this.
- Project Objective: Develop a search engine using an inverted index for literary texts sourced from the Gutenberg Project using Java Programming Language.
- Challenges: Efficient data collection, scalable indexing, and query optimization in terms of speed and memory management.

Project Modules

Web Crawler

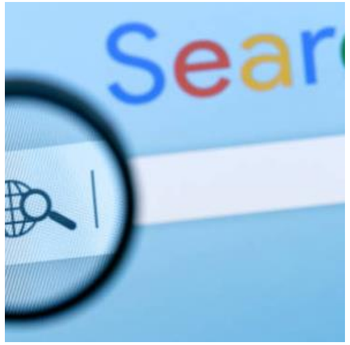
Automatically collects literary works from the Gutenberg Project.

Inverted Index

Stores each word along with its position in the documents.

API

Allows to use the functionality of search engine using HTTP queries



Query Engine

Allows users to search for words, returning results with the specific locations of the words in the documents.

Inverted Index Data Structures

Trie Node

-Efficient for searches involving common prefixes.

-Disadvantages: Higher memory usage if words do not share many prefixes.

-Implementation: Words are inserted character by character, with each path representing a word. A special marker indicates the end of a valid word.

-Persistence: The Trie is serialized to JSON format for future use.

Hash Index

-Each word is assigned to a bucket based on its hash value, allowing for quick retrieval.

-Advantages: Quick lookups with constant time complexity.

-Concurrency: Parallel processing is used to handle large data efficiently.

-Collision Handling: Multiple words with the same hash are handled using lists in the same bucket.

Inverted Index Data Structures

File Per Word Index

-Each file contains only one word so no need to load the entire index

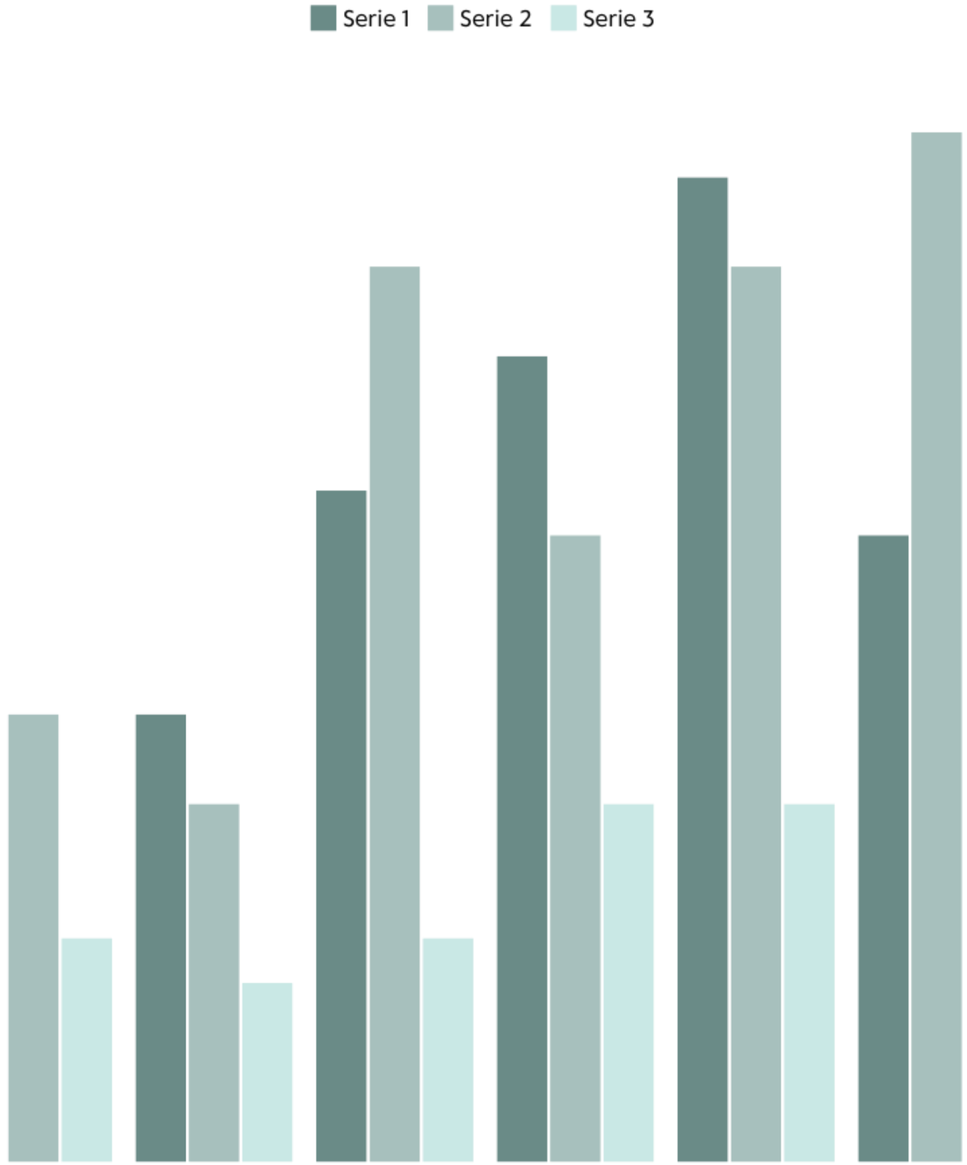
-Advantages: Optimized for inserting since is not dependent of how big the index already is.

-Concurrency: Parallel processing can be used in the next stage to handle large data efficiently.

-Query efficiency: searching for any word takes O(1) as the file where the word is stored has the name of that word.

BenchMark

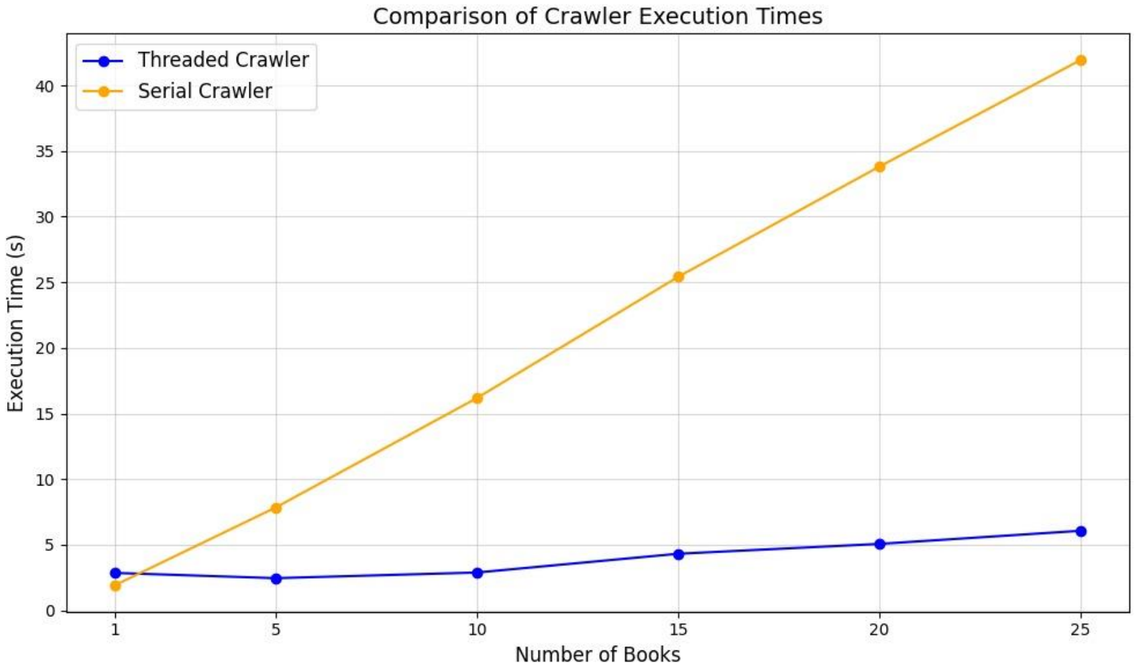
We tested all the implementations of the inverted index to test which one of them gives a better performance.



Results Crawler

Benchmarking of the Indices

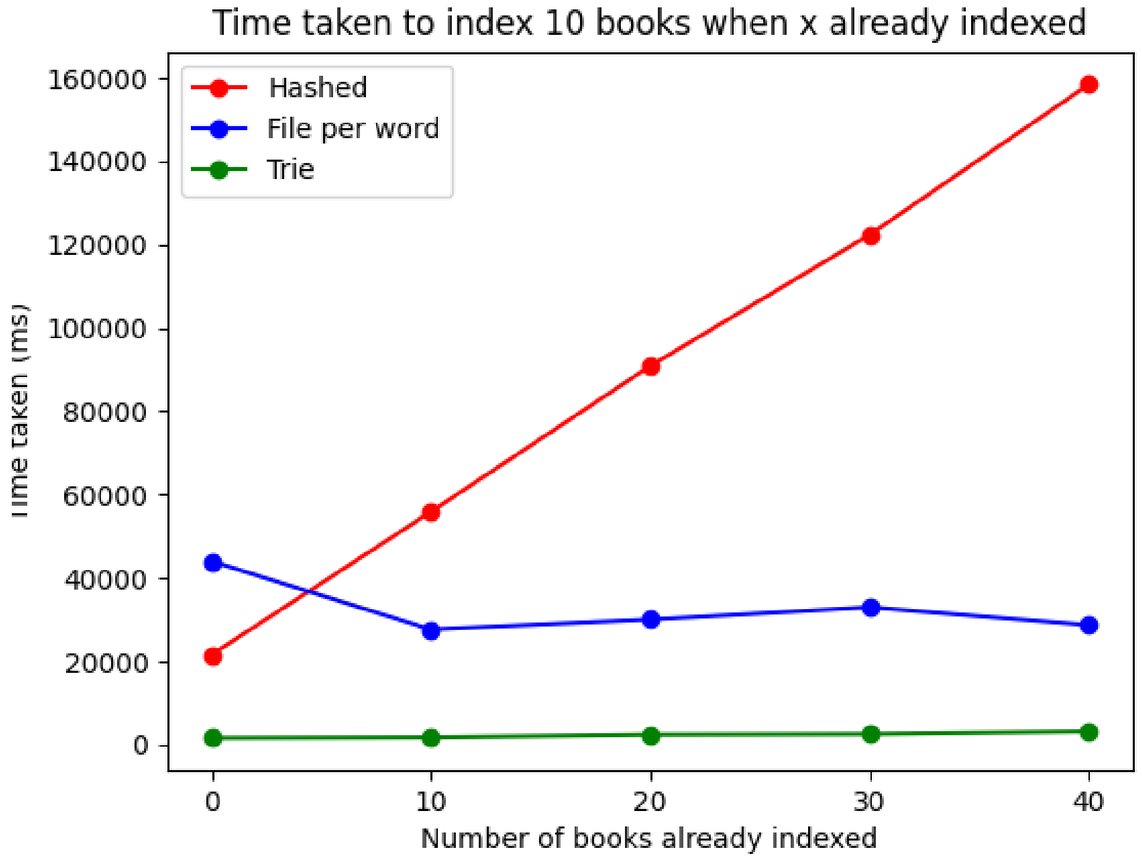
1. Serial approach:
   - The time of crawling increases linearly
   - Big slope
2. Parallel approach:
   - Also increases linearly.
   - But its slope is less.



Results

Benchmarking of the Indices

1. Trie Index:
   - Stable over the time.
   - High Performance (3 seconds for 10 books +/-).
2. Hashed Index:
   - Linearly dependent on the number of books already indexed.
   - Very poor performance.
3. One File Per Word Index:
   - Independent of the number of books already indexed.
   - Good performance, specially for querying.



Results

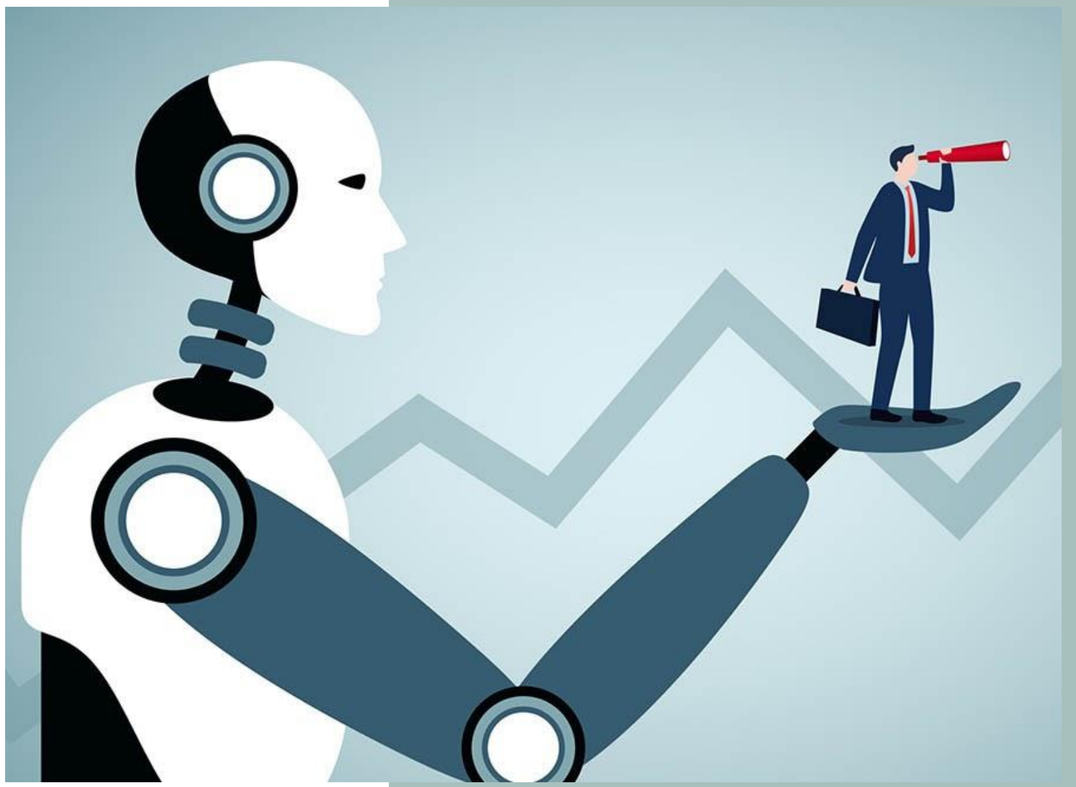
- Data retrieval performance tested across all indexes using JMH with consistent search terms
- File-per-word index delivered the best results with response times under 1 ms
- Trie index ranked second, followed by hashed index, both taking several seconds to search

Structure /Test	Hashed	Trie	File Per Word
One Word Search	1938.594	1021.161	0.174
Two Word Search	6840.687	4067.887	0.270

Structure /Test	Hashed	Trie	File Per Word
One Word Search (With Filter)	2094.830	1323.391	0.243
Two Word Search (With Filter)	8821.508	4702.541	0.278

Conclusion and Future Work

- The Trie Index outperforms the Hashed structure in terms of speed and consistency, making it the superior choice for large-scale applications. One file per word index shows acceptable performance.
- One file per word index is very efficient for searching, the hashed implementation and the trie index not that much.
- The one file per word index can be optimized by taking advantage of parallelization (using threads).



Thanks for your atention!