# **Project Phase 1: Data Structure Design and Implementation**

Jacob Jeffers

MSCS 532

Dr. Vanessa Cooper

Fall 2024

11/3/2024

Project Phase 1: Data Structure Design and Implementation

This project aims to create a basic yet efficient search engine using essential data structures that handle large-scale text queries. The project will involve indexing a collection of documents, responding to keyword-based queries, and returning relevant results quickly. After reviewing several options, I built a search engine to provide a rich learning experience for implementing and optimizing data structures. I also thought studying how Google and Bing perform their magic would be interesting. The search engine revolutionized the internet world, and I think it would be so fun to create my version of one from scratch.

I selected two core data structures to achieve these objectives: an inverted index and a trie for supporting autocomplete functionality. The inverted index forms the backbone of the search engine, acting as a dictionary where each unique word maps to a list of document IDs in which it appears (GeeksforGeeks, 2024). This setup allows for rapid lookups, as each keyword in a search query can directly retrieve relevant document references from the index. Implementing this structure in Python involves processing a collection of documents, tokenizing each document into individual words, and constructing a dictionary where each word points to the corresponding document IDs. This design ensures the search engine can efficiently access documents containing the queried terms, providing the basis for effective search operations.

The trie (or prefix tree) further enhances the search engine by supporting prefix-based searches and potential autocomplete functionality (Kalaydjian, 2023). This tree-like structure enables quick retrieval of words with common prefixes, improving the search experience by suggesting relevant terms as the user types. The trie is constructed with nodes representing individual characters, allowing for efficient lookups based on word prefixes and improving the engine's usability. After implementing these data structures, I will conduct performance tests to measure response times and memory usage.

**Coding and Testing of the Inverted Index and Trie Data Structures**

I created two essential data structures to implement the search engine's core functionalities: an inverted index for efficient document retrieval by keyword and a trie to support optional prefix-based searching or autocomplete features. The coding and testing of these structures were focused on ensuring their correctness, efficiency, and scalability for larger datasets.

***Inverted Index Implementation***

The inverted index forms the backbone of the search engine, allowing for quick lookups of documents containing specific keywords (GeeksforGeeks, 2024). I implemented this structure using Python's defaultdict, where each word is mapped to a list of document IDs representing the documents in which it appears. This structure enables efficient retrieval of document references, as each query term directly points to the relevant document IDs in the dictionary. The add\_document method tokenizes each document into individual words and then updates the index by adding the document ID to each word entry. I used a set to avoid duplicates, ensuring each document ID appears only once per word in the index. Additionally, the search method allows for quick retrieval of documents for a given query.

To test the inverted index, I created sample documents and ran searches for various keywords to validate its functionality. The tests confirmed that each query correctly returned the appropriate document IDs. I also designed test cases to handle edge cases, such as words that did not appear in documents with repeated words. Overall, the inverted index performed accurately across these tests and had a quick response time of 0.083s, demonstrating its suitability for handling text-based queries in the search engine.

A screenshot of a computer screen

Description automatically generated

A black screen with orange text

Description automatically generated

***Trie Implementation for Prefix-Based Search***

I implemented a trie (or prefix tree) to support prefix-based suggestions and autocomplete for enhanced search capabilities (Kalaydjian, 2023). The trie is structured with nodes representing characters, enabling efficient lookups based on word prefixes. I defined a TrieNode class to represent each node and a Trie class containing methods for inserting words and searching prefixes. This structure allows the search engine to quickly retrieve words that share common prefixes, which can improve the user experience by suggesting relevant terms as users type.

I tested the trie by inserting sample words and conducting prefix searches. For instance, prefix searches such as "pro" and "pyth" accurately returned matches, while prefixes like "java," which were not in the trie, returned no results. This testing confirmed the trie's effectiveness in supporting prefix-based searching. It was also relatively fast when combined with the inverted index, producing results of 0.121 s.

A screenshot of a computer screen

Description automatically generated

A black screen with white text

Description automatically generated

***Efficiency and Performance Testing***

According to Tunkelang (2022), evaluation should be at the heart of the development process. To improve the design for future testing and implementation, I conducted performance tests to assess each data structure's time and memory size. I tested the inverted index and trie it on larger datasets by generating a collection of random documents with hundreds of words. These tests measured the time required for both document insertion and query processing and the memory footprint of each data structure. The results showed that the inverted index scaled efficiently for document additions and searches, while the trie maintained fast prefix lookups even as the dataset grew. The inverted index time to add 1000 documents was 0.0181s with a memory size of 1.92 million bytes. The time taken to trie 1000 documents was 0.1052s with a memory size of 48 bytes.

A screenshot of a computer program

Description automatically generated

**Conclusion**

The initial performance metrics indicate that the chosen data structures support the search engine's requirements, with both structures showing accuracy and scalability. In phase one of this project, I have created a foundation for a high-performance, scalable search engine tailored to handle complex text-based queries by implementing and optimizing these data structures. The next phase of this project will look at developing a Proof of Concept (PoC) to demonstrate functionality. It will highlight the inverted index and trie functions to set a foundation for further development.

GitHub Repository for Phase 1 Code: <https://github.com/jakejeffers/ProjectPhase1>

References

GeeksforGeeks. (2024, March 11). *Inverted Index*. GeeksforGeeks. <https://www.geeksforgeeks.org/inverted-index/>

Kalaydjian, M. (2023, December 25). I am building a Search Engine using a Trie Data Structure. *Medium*. <https://medium.com/@maxi.gkd/building-a-search-engine-using-a-trie-data-structure-cb79475d8a3d>

Tunkelang, D. (2022, January 20). Measuring search effectiveness - Daniel Tunkelang - Medium. *Medium*. <https://dtunkelang.medium.com/measuring-search-effectiveness-a320bd6bdd7a>