# **MSCS 532 Search Engine Project**

Jacob Jeffers

MSCS 532

Dr. Vanessa Cooper

Fall 2024

11/24/2024

**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, the developer built a search engine to provide a rich learning experience for implementing and optimizing data structures. Studying how Google and Bing perform their magic would be interesting. The search engine revolutionized the internet world, and the developer thinks it would be so fun to create their version of one from scratch.

To start, the developer 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**

The developer 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). The developer 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. The developer also 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.

The developer created sample documents to test the inverted index and ran searches for various keywords to validate its functionality. The tests confirmed that each query correctly returned the appropriate document IDs. The developer 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***

The developer 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 (Fredkin, 1960). The developer 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.

The developer 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, the developer conducted performance tests to assess each data structure's time and memory size. The developer 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 of Phase One**

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, the developer 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>

**Project Phase 2: Proof of Concept Implementation**

For phase two of this project, the developer will demonstrate the core functionality of the inverted index and trie data structures. This will be done through a series of code implementations (tasks).

**Task 1: Partial Implementation of Data Structures**

The first task involves revisiting and refining the existing code to handle insertion, searching, and error handling. The developer added modular methods for the inverted index code to handle document insertion and keyword searching. The trie data structure was expanded to include an autocomplete function. A screengrab of the proposed code can be found below.

A screen shot of a computer program

Description automatically generated

**Task 2: Demonstration of Key Operations**

Task 2 is the main task for this project's proof of concept (PoC). It involves showcasing the functions of the chosen data structures. The developer will implement a script that demonstrates adding, searching, and displaying test cases to validate functionality.

The PoC code implementation can be found below. The code demonstrates how the data structures support the search engine by performing basic searches. It initializes both data structures in the previous code and then generates sample documents filled with randomly generated words. For each document, the inverted index stores each work alongside the document's ID number, while the trie stores each word by character, allowing fast autocomplete lookups. To showcase the functionality, the code searches the inverted index for a specific word and then performs a prefix search on the trie. This returns a list of words that match the prefix.

A screen shot of a computer program

Description automatically generated

An example output of the code can be found below.

A screen shot of a computer code

Description automatically generated

**Task 3: Documentation of Implementation Process**

During code development, the most significant challenges faced included efficiently handling duplicate words in the inverted index and memory use in the trie. It took hours to figure out how to handle redundant entries effectively. To address this issue, data storage was prioritized by sets to avoid duplicate document ID numbers. The code design process made each data structure flexible and easy to identify and extend upon. For example, the inverted index code includes separate methods like add\_document for adding documents, search for querying by keyword, and delete\_document for removing documents. This approach will make it easier to add or adjust existing code in the future.

**Looking Ahead**

Future improvements could involve scaling the PoC to assess how the structures perform with large datasets. The developer is also considering adding new features, like wildcard searches in the trie or more complex query handling in the inverted index. To complete the full implementation, the code will need extended search capabilities, performance optimization, the development of a user interface, and user testing to help identify any issues before a final release.

GitHub Repository for Phase 2 Code: <https://github.com/jakejeffers/MSCS532-Project>

**Project Phase 3: Optimization, Scaling, and Final Evaluation**

In this phase, the developer will expand on the proof-of-concept implementation from the previous phase by providing additional optimizations for performance and scalability. In addition, the developer will test the optimizations and analyze the results. This phase aims to refine the implementation to handle more complex scenarios and larger datasets, ensuring that the code is efficient and ready for real-world applications.

**Optimization Techniques**

*Inverted Index*

To enhance the inverted index, query caching will be implemented to store frequently searched terms and their corresponding results. This avoids redundant computations for repeated queries, significantly reducing query response time (Lindsley, 2023). Additionally, compression techniques (e.g., gap encoding) will be applied to reduce the storage size of the document ID lists.

A screen shot of a computer program

Description automatically generated

*Trie*

The trie will be converted to a compressed trie (or radix tree) to optimize storage by combining nodes with common prefixes. This reduces memory usage while maintaining the ability to perform fast prefix-base d lookups (Ankit, 2024).

A screenshot of a computer program

Description automatically generated

**Scaling Strategy**

In the previous phase for the inverted index, the developer had used a list to store document IDs, but the code implemented in phase three stored document IDs into a set. This allowed for faster add, delete, and search operations. For the trie, it used manual initialization, but the current code uses defaultdict, which optimizes the node creation. To stress-test the code, the new version processes 10,000 documents with 100 words each to demonstrate scalability through timing and memory usage metrics.

**Testing and Validation**

The developer took timing and memory usage metrics for each data structure across various operations to test the implemented code. The table below summarizes the results and explains them.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Operation** | **Structure** | | **Time Taken** | **Memory Usage** | **Explanation** |
| Add Documents | | Inverted Index | 0.7913s | 30.7 MB | Efficiently stores document IDs for words; faster insertion due to set-based storage. |
| Add Documents | | Trie | 4.8174s | 48 B | Slower insertion as each word is processed character by character. Memory here excludes full nodes. |
| Search Query | | Inverted Index | 0.0000s | N/A | Quick lookup using hashed keys; scales well with large datasets. |
| Search Query | | Trie | 0.0001s | N/A | Character-by-character traversal for prefix queries remains fast, even with 1,000,000 words. |
| Memory Usage (Total) | | Inverted Index | N/A | 30.7 MB | Handles large datasets efficiently with moderate memory usage. |
| Memory Usage (Total) | | Trie | N/A | 48 B | Shows minimal root memory usage; total node memory usage needs advanced profiling. |

**Performance and Analysis**

The results show how the Inverted Index and Trie handle performance and scalability differently. The Inverted Index is much faster for adding documents and uses moderate memory, which makes it great for direct word searches. The Trie, while slower to build because it processes words one character at a time, shines in prefix-based searches and handles larger datasets without losing speed. Its memory usage at the root level is minimal, though profiling the entire structure would give a better picture. Overall, both structures scale well and are optimized for specific tasks, making them ready for handling more significant, real-world scenarios.

**Real-World Applications**

The work done in this project has precise, practical applications. An optimized inverted index and trie can power systems like search engines, document retrieval systems, and autocomplete features. For example, the inverted index could quickly search through massive datasets, like legal documents or academic papers, by efficiently mapping keywords to relevant files. On the other hand, the trie's prefix-based search capability makes it ideal for real-time autocomplete in applications like search bars or text prediction tools. By scaling these data structures to handle large datasets, this project lays the groundwork for integrating these concepts into real-world systems that demand speed, accuracy, and scalability.

GitHub Repository for Phase 3: <https://github.com/jakejeffers/MSCS532-Project>

**References**

Ankit. (2024, November 16). An introduction of Radix Tree - Ankit - medium. *Medium*. <https://ankitpandeycu.medium.com/unleashing-the-potential-of-radix-tree-35e6c5d3b49d>

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>

Lindsley, C. (2023, October 21). The Power of Caching: An introduction to caching within Python/SQL code. *Medium*. <https://medium.com/@55swimmer55/the-power-of-caching-an-introduction-to-caching-within-python-sql-code-5cc29af4803d>

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