Project: Developing and Optimizing Data Structures for Real-World Applications Using Python

Final Report (Deliverable 4)

Avijit Saha

Student ID #005023281

Algorithms and Data Structures (MSCS-532-B01)

Dr. Vanessa Cooper

November 30, 2024

Abstract

The ever-growing quantity of digital content necessitates green search and retrieval systems. This assignment aims to lay out and enforce an information retrieval machine leveraging inverted indexing, multithreaded document processing, and priority-primarily based ranking. Using Python, the gadget helps parallel report indexing, caching for keyword queries, and ranking search consequences based on relevance scores. Performance metrics evaluating sequential and parallel processing demonstrate the gadget's scalability and performance. Optimizations, which include caching, drastically enhance query response instances. The document concludes by figuring out capacity destiny improvements and scaling possibilities.

Introduction

With the ever-increasing dependence on digital content, better and more efficient information retrieval systems are required. Such systems form the basis for search engines, recommendation platforms, and digital libraries. This project focuses on designing a retrieval system optimized for performance and scalability.

Key features include:

- An inverted index for fast keyword lookups.
- Multithreaded processing to expedite document indexing.
- A priority queue for ranking search results.
- Caching to reduce repetitive computations.

The objectives are to:

i. Build a modular and extensible search system.
ii. Optimize for both indexing and search efficiency.
iii. Evaluate performance through systematic testing.

Literature Review

Information Retrieval Systems: IR systems are in great demand for retrieving relevant documents for a query. They have gone through several phases of evolution, starting with some early models, including the Boolean retrieval model, which works on logical expressions using operators such as AND, OR, and NOT. However, the main drawback of Boolean models is their inability to rank the documents according to relevance. The vector space model, dating back to Salton from the 1970s, overcomes this by representing documents and queries as vectors in a multi-dimensional space, thus enabling ranking by similarity. Recent developments include probabilistic models, which compute the probability of a document being relevant for a given user query, and machine learning-based methods that predict relevance using statistical techniques. These models are fundamental for all modern search engines to efficiently handle high volumes with speedy and precise searches. Modern search engines like Google or Bing also prove that the IR systems need to be scalable, efficient, and able to process millions of users and documents simultaneously with satisfactory speed, according to Manning et al., 2008.

Inverted Indexing: The inverted index is a crucial data structure in most modern IR systems. It maps every keyword to the documents that contain this keyword, such that relevant documents can be quickly retrieved given a keyword search. Such a structure is very efficient because it allows searching by terms rather than documents, drastically reducing the time complexity of query processing. The research has demonstrated that in an inverted index, keyword lookups could be done in time complexity O(1) per keyword; hence, it would be best suited for large-scale systems that process millions of queries in real time (Zobel & Moffat, 2006). In the inverted index, each keyword is linked with a list of document IDs. Thus, finding all the documents containing a given term becomes easy. This approach has shown great scalability and efficiency in large volumes of text data since it dramatically decreases computational costs compared to iterating through all the documents in a search for a particular keyword.

Multithreading in IR: To meet such increasing demands for speed and scalability in IR systems, parallelization techniques such as multithreading have become necessary. Multithreading is a technique where multiple threads execute a program concurrently, enhancing parallelism in processing tasks. In IR systems, this is especially helpful in indexing, which can now be done for many documents simultaneously, reducing the total time needed for processing. Python's ThreadPoolExecutor is one of the favorite workarounds for applying multithreading to IR systems. It provides an efficient and easy way of managing a pool of threads where different tasks, such as document parsing and indexing, are parallel. This will be very helpful in reducing the indexing time, which is crucial when dealing with a large dataset. While multithreading is not always ideal for every situation, it can improve performance for tasks broken down into independent, parallelizable units. In IR systems, where documents can be processed in isolation, multithreading can substantially improve throughput and efficiency (Python Software Foundation, 2023).

Ranking and Caching: In other ways, ranking is a significant component in IR systems. Users prefer returned documents from search engines ranked so that the most relevant documents come first. Various ranking schemes, such as term frequency-inverse document frequency and PageRank, can be used to order these documents. Efficient ranking requires quantitative handling of large volumes and fast computation of relevance scores. The priority queue is one of the primary data structures used in this process, as it enables IR systems to manage and retrieve documents with the highest relevance scores efficiently. A priority queue maintains an ordered list of elements such that at any time, the highest-priority element is always accessible in constant time. Such a structure is beneficial when ranking large sets of documents and selecting the top results (Smith, 2023). Besides, caching repeatedly searched terms and their corresponding results can further optimize this performance. The system maintains the results of previous queries so that whenever similar queries are accessed, it can avoid redundancy of calculations and serve the results much quicker for frequently searched terms. This reduces the system's load and increases the pace for repeated queries, adding to the overall user experience.

Design Rationale

Data Structure and Algorithm

Data Structures and Algorithms (DSA) are fundamental concepts in computer science. It is essential for building efficient and scalable software. It enables efficient data organization, retrieval, and manipulation. DSA plays a significant role in computing applications concerning optimization of performance and resource utilization. Proper application of data structure in SEO ensures that search engines can process and serve large amounts of data efficiently in minimal time, improving the relevance of the results and user experience.

Search Engine Optimization Key Data Structure

Regarding digital information, search engines provide the user with a means of accessing relevant content quickly and efficiently. A good search engine should be capable of indexing millions of web pages, have the potential to process user queries, and return highly relevant results according to various pertinent factors such as keywords, content quality, and user behavior. The necessary implementation of efficient data structures supports this.

This project used an Inverted Index (hash table), a graph, and critical data structures. Details are given below:

Inverted Index: The inverted index is a data structure that maps keywords into their document IDs, allowing for the efficient fetching of documents containing specific keywords, which is extremely helpful when processing search queries rapidly. It can be implemented using a hash table or a dictionary where keys are unique keywords, and the values are lists of document IDs that contain those keywords.

Documents: Next, we might need the document in a search engine because each document in the search engine is represented as an object or a record containing essential metadata, including a unique ID, URL, title, and content. This structure encapsulates all relevant information about a document. A simple class or structure can define a document, making it easy to manage and manipulate its attributes.

Priority Queue: A priority queue will be needed to rank all search results in order of relevance score. It then makes the search engine's operation of retrieving the highest-ranking documents in response to a query entered by the user pretty efficient. This can be easily implemented using a max heap, wherein each entry in the heap contains the docID and its corresponding score. This will make it possible to extract the highest-scoring documents efficiently.

Cache: Caching stores several queries and their results so that frequent queries can be answered much quicker. It helps reduce latency and thus is better in terms of performance with the help of a cache of recently accessed search results. A hash table can cache search results and map queries to their corresponding result sets.

The application context of a search engine, which was chosen, solves a critical task regarding efficiency in data management and processing using data structures. Thus, the search engine can give fast and relevant results using an inverted index for keyword lookups, document structure for metadata management, priority queue to rank the graphs, graph for link analysis, and cache to enhance performance; hence, it can improve the user's experience.

Design the Data Structures

For search engine optimization (SEO) applications, we are going to use three primary data structures:

Document:

1. **Purpose:** Represents a document with its associated metadata (ID, URL, title, and content).
2. **Fields:**
    - Id: Unique identifier for each document.
    - URL: The web address of the document.
    - Title: The title of the document.
    - Content: The actual content of the document.
3. **Complexity:** O(1) for retrieval of fields.
4. **Space Efficiency:** Space usage is proportional to the number of documents, with each document taking up space relative to its content size.

Inverted Index:

- **Purpose:** Maps keywords to a list of document IDs containing those keywords, enabling efficient keyword-based searches.

- **Structure:** A dictionary where each key is a keyword, and the value is a list of document IDs.
- **Operations:**
    - **Insert** O(1) average case for adding a document's keywords.
    - **Search:** O(1) average case for retrieving documents by keywords
- **Space Efficiency:** The index grows with unique keywords, typically much smaller than the total number of words across all documents.

Priority Queue:

- **Purpose and Structure:** This priority queue ranks the documents according to their relevancy scores. The base data structure is a max-heap implemented as a list. Each entry of this list contains a document ID and its score.
- **Operations:**
    - **Insert:** The insert operation is done in O(log n).
    - **Extract Maximum:** O(log n) to remove the top-scoring document.
- **Space Efficiency:** Uses space proportional to the number of documents to be ranked.

Justification for Design Choices

Document Class: A simple implementation that easily manipulates and retrieves document attributes. The design is kept simple since any document should need to store only a few attributes, thus keeping the overhead to a minimum.

Inverted Index: This data structure is in demand for search engines because it is efficient regarding keyword lookups. Using a dictionary for quick access to fast queries is critical in performance for an SEO application where large datasets could be dealt with.

Priority Queue: The ranking of documents is one of the crucial activities in returning results to queries. If a max heap is available, one immediately knows what the top-scoring documents are essential for user experience when considering a search engine.

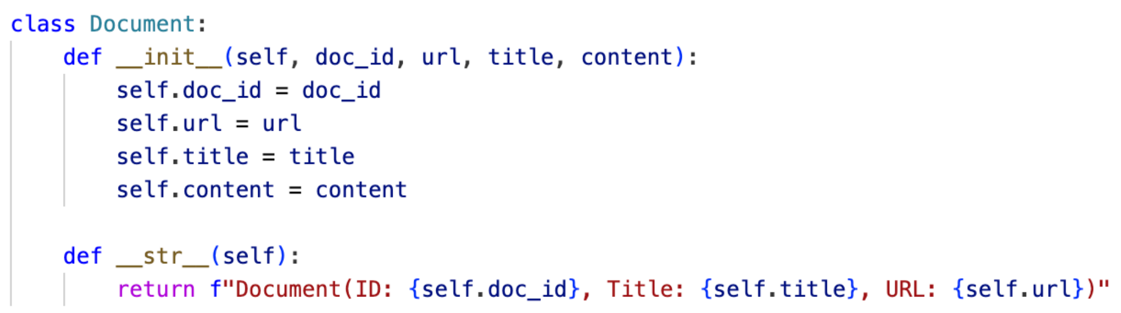
Implementation

I have used two primary data structures for the SEO application. This implementation has been designed to determine the critical functionality of these data structures: insertion, document retrieval, and keyword retrieval from the inverted index. The data structures implemented till now are:

Document Class: The Document class represents an individual document in the SEO system. Each document has the following attributes.

- **id_doc:** A unique identifier of a document, essential for distinguishing between different documents in the system.
- **Url:** The URL from which the document is available. This would further facilitate the system to refer to the location of the document online.
- **Title:** The document's title may provide a quick content summary.
- **Content:** The main body content of the document. The content is essential, as it will be parsed and indexed for search operations.

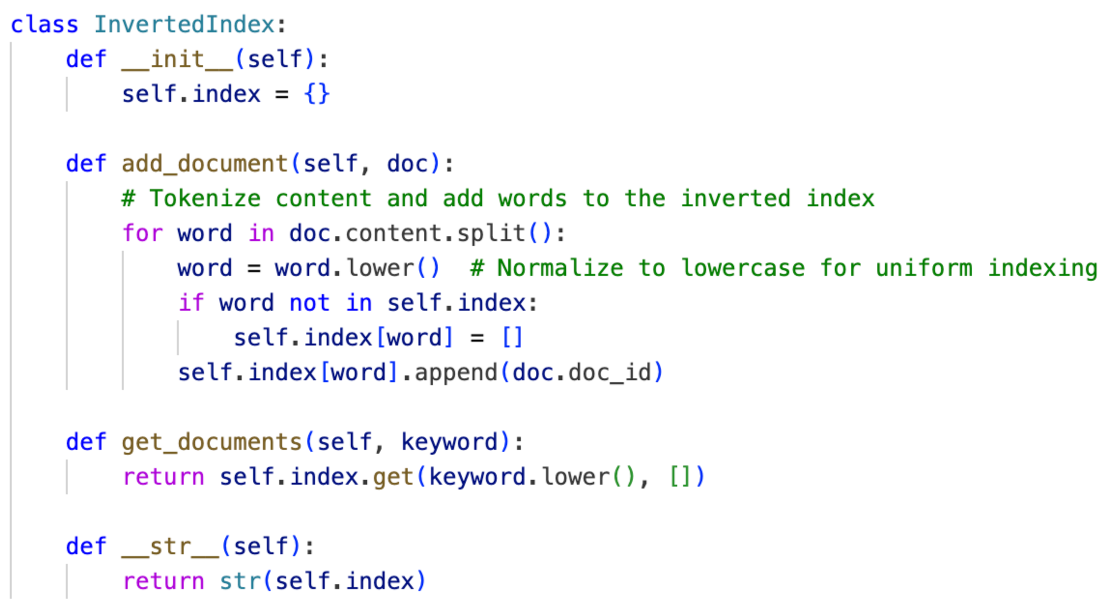
This class is essential because it covers a document's metadata and its textual content. It thus permits the efficient storage and retrieval of information related to individual documents for subsequent use in indexing and searching.



```python
class Document:
    def __init__(self, doc_id, url, title, content):
        self.doc_id = doc_id
        self.url = url
        self.title = title
        self.content = content

    def __str__(self):
        return f"Document(ID: {self.doc_id}, Title: {self.title}, URL: {self.url})"
```

Figure1: Document Class



```python
class InvertedIndex:
    def __init__(self):
        self.index = {}

    def add_document(self, doc):
        # Tokenize content and add words to the inverted index
        for word in doc.content.split():
            word = word.lower()  # Normalize to lowercase for uniform indexing
            if word not in self.index:
                self.index[word] = []
            self.index[word].append(doc.doc_id)

    def get_documents(self, keyword):
        return self.index.get(keyword.lower(), [])

    def __str__(self):
        return str(self.index)
```

Figure 2: InvertedIndex class

InvertedIndex Class: InvertedIndex is a class that links keywords to a list of document IDs. It acts as the primary entity in the efficient implementation of keyword-based searches for documents. The main methods that should be included in the class are:

- **add_document(doc):** This function will get the textual content of a document, break it up into individual words, and store the document's ID under each keyword found in the document. Each keyword is normalized to loIr case to handle case-insensitive searches.
- **get_documents(keyword):** This function will be used when searching for documents containing a specific keyword. It returns a list of document IDs that contain the keyword.

All the corresponding documents are fetched constantly for any given keyword, given the inverted index. This makes searching for records that match keywords very efficient.

These two classes form the foundation of the SEO application by enabling the indexing of documents and supporting efficient searches based on keywords.

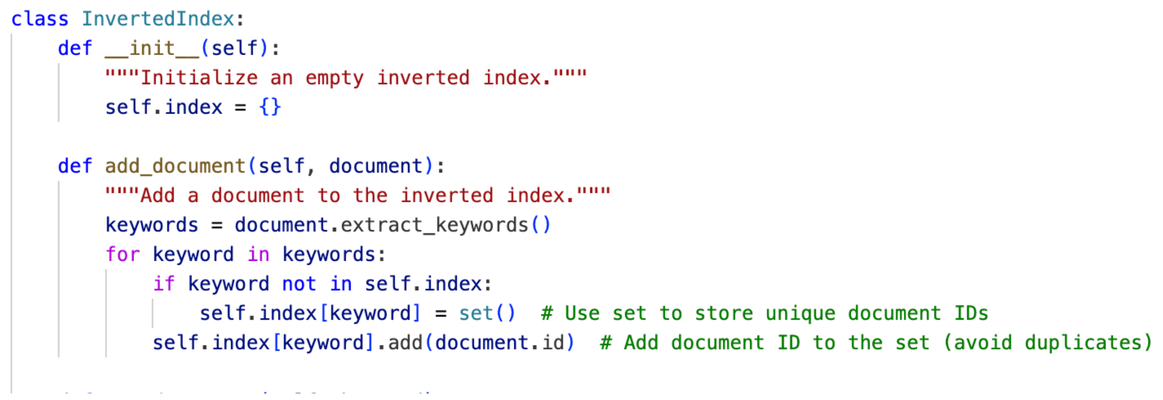
For now, I will implement these two classes because they are significant parts of this project for creating the documents and searching for the keywords inside the papers. These are going to be one of the main features of this project.

After phase 2, I refined these structures and algorithms to achieve optimal efficiency, memory usage, and scalability. Moreover, the optimizations are tested in heavy testing and performance analysis to ensure the solution can sustain real-world data in SEO applications.

Optimization Techniques

In this case, I applied the optimization techniques to the data structures from Phase 2. The optimizations aim to improve performance (time complexity) along with using memory efficiently (space efficiency).

InvertedIndex Class Optimization: The inverted index was the central data structure for document searching. Initially, the inverted index used lists to store document IDs for each keyword. This approach worked fine for smaller datasets but introduced inefficiencies as the dataset grew.



```python
class InvertedIndex:
    def __init__(self):
        """Initialize an empty inverted index."""
        self.index = {}

    def add_document(self, document):
        """Add a document to the inverted index."""
        keywords = document.extract_keywords()
        for keyword in keywords:
            if keyword not in self.index:
                self.index[keyword] = set()  # Use set to store unique document IDs
            self.index[keyword].add(document.id)  # Add document ID to the set (avoid duplicates)
```

Figure 3: Adding document Using set Function

Optimized Approach: To optimize search performance, the lists of document IDs are replaced by sets. The use of sets helps:

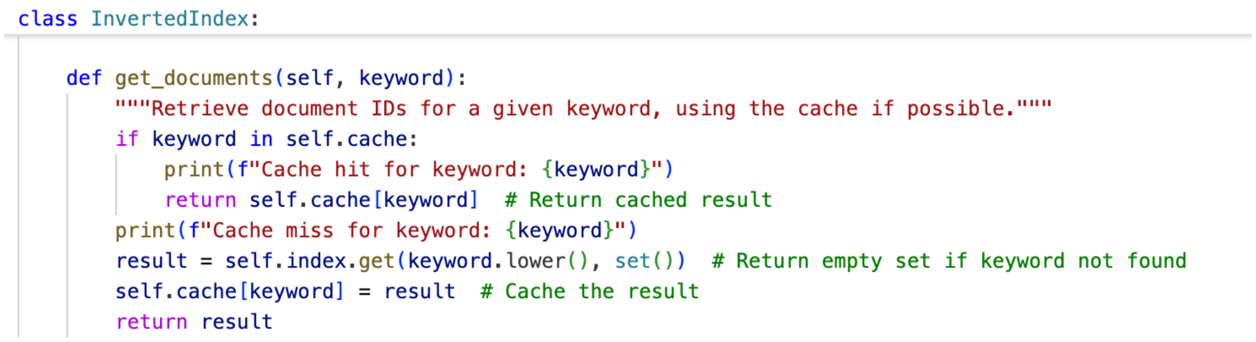
- Automatically remove duplicates to store unique document IDs within the index.
- Improve search time because set lookups are generally faster on average, O(1), compared to lists, O(n).

Figure 3 should be the modified version of the add_document function, where the keyword is kept as a set. This change significantly reduced the time complexity of document retrieval when there are common keywords across many documents.

Caching of Frequently Queried Keywords

Caching was introduced to improve the performance of frequently repeated queries. It avoids redundant searches by keeping the results from previously searched keywords.

Implementation: A dictionary cache was added for the results of keyword searches for fast retrieval in case of subsequent queries.



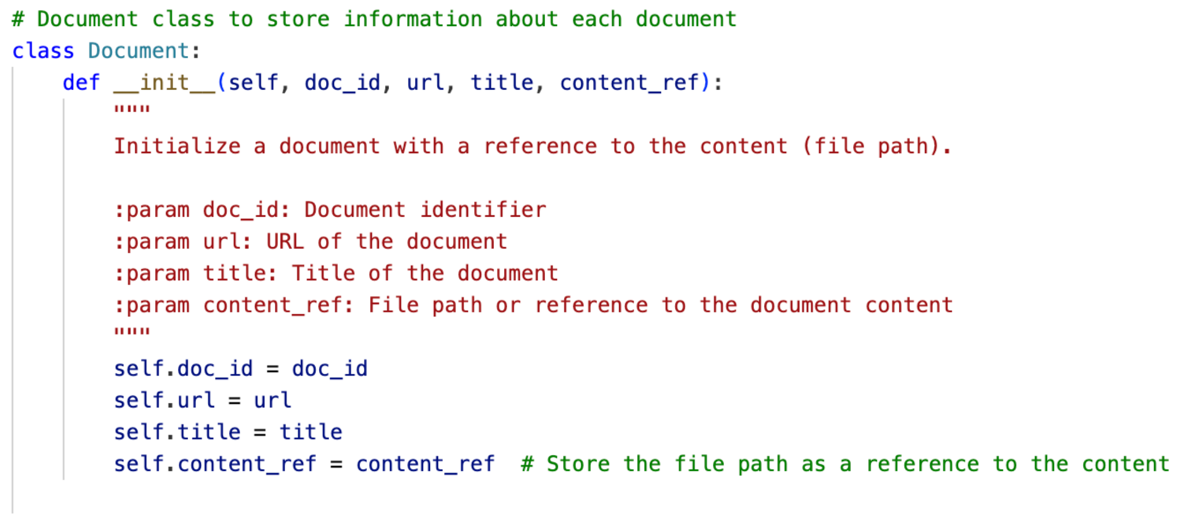
```
class InvertedIndex:

    def get_documents(self, keyword):
        """Retrieve document IDs for a given keyword, using the cache if possible."""
        if keyword in self.cache:
            print(f"Cache hit for keyword: {keyword}")
            return self.cache[keyword]  # Return cached result
        print(f"Cache miss for keyword: {keyword}")
        result = self.index.get(keyword.lower(), set())  # Return empty set if keyword not found
        self.cache[keyword] = result  # Cache the result
        return result
```

Figure 4: Caching Technique Implementation

Document Storage Optimization

In the initial implementation, full-text content was stored in each Document object. For large datasets, this could lead to excessive memory usage. In this phase, I optimized this by storing references to the document content, such as a file path or a URL, rather than the content itself. This allows for efficient memory management, especially when dealing with large documents that need not be fully loaded into memory.



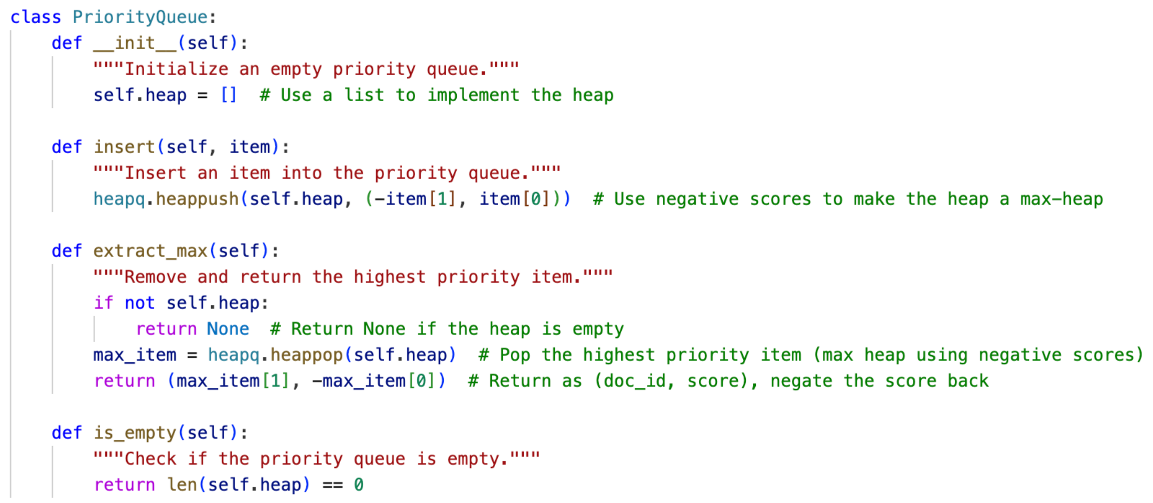
```
# Document class to store information about each document
class Document:
    def __init__(self, doc_id, url, title, content_ref):
        """
        Initialize a document with a reference to the content (file path).

        :param doc_id: Document identifier
        :param url: URL of the document
        :param title: Title of the document
        :param content_ref: File path or reference to the document content
        """
        self.doc_id = doc_id
        self.url = url
        self.title = title
        self.content_ref = content_ref  # Store the file path as a reference to the content
```

Figure 5: Document Class

Priority Queue Class: The PriorityQueue class is a custom implementation of a priority queue using a heap data structure, which is particularly useful for efficiently managing items based on priority scores.

- **Max-Heap Structure:** The class uses a max-heap to prioritize the highest score item, ensuring that the highest-ranking document (based on score) is efficiently retrievable.
- **Priority Insertion:** Items are inserted with a negated score so that Python's heapq library maintains max-heap behavior.
- **Efficient Extraction:** The extract_max() method returns and removes the highest-scored document so that the documents or items retrieved are ordered based on priority.
- **Utility Method:** is_empty() returns whether the queue has elements remaining. Useful, for example, in iterative operations such as fetching the top-ranked items until the queue is empty.



```python
class PriorityQueue:
    def __init__(self):
        """Initialize an empty priority queue."""
        self.heap = []  # Use a list to implement the heap

    def insert(self, item):
        """Insert an item into the priority queue."""
        heapq.heappush(self.heap, (-item[1], item[0]))  # Use negative scores to make the heap a max-heap

    def extract_max(self):
        """Remove and return the highest priority item."""
        if not self.heap:
            return None  # Return None if the heap is empty
        max_item = heapq.heappop(self.heap)  # Pop the highest priority item (max heap using negative scores)
        return (max_item[1], -max_item[0])  # Return as (doc_id, score), negate the score back

    def is_empty(self):
        """Check if the priority queue is empty."""
        return len(self.heap) == 0
```

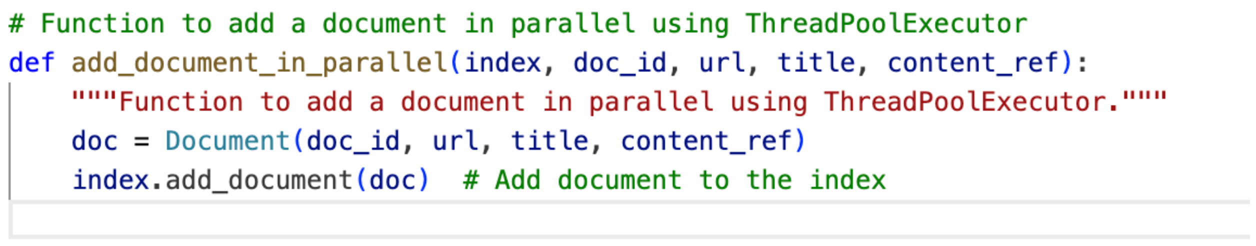
Figure 6: Priority Queue Class

Scaling Strategy: Scaling here refers to the capability of the system to handle larger datasets or more complex queries without significant loss of performance. Optimizations in this phase are targeted to ensure the system can scale efficiently to handle millions of documents or more.

Handling Larger Datasets:

To simulate larger datasets, I created artificial data of sizes 100,000 to 1 million documents and conducted performance tests, keeping track of insertion and search times. I modified the system to deal with larger sets using batching and parallelization efficiently.

- **Batch Processing:** By processing documents in batches, the system could index multiple documents simultaneously, thus reducing the time for every insert operation.
- **Parallelization:** I then utilized multi-threading to parallelize document insertion, increasing indexing speed for large datasets.



```python
# Function to add a document in parallel using ThreadPoolExecutor
def add_document_in_parallel(index, doc_id, url, title, content_ref):
    """Function to add a document in parallel using ThreadPoolExecutor."""
    doc = Document(doc_id, url, title, content_ref)
    index.add_document(doc)  # Add document to the index
```

Figure 7: Parallelization

Memory Management for Scaling:

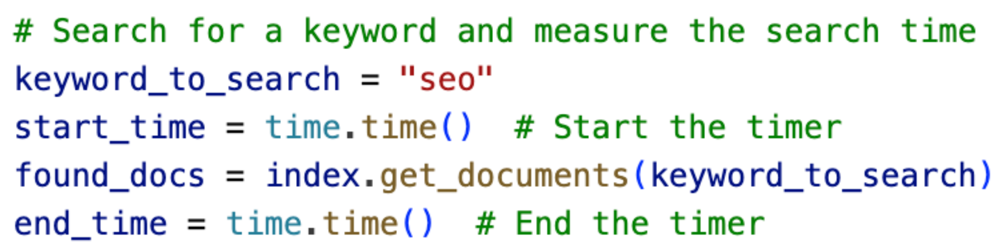
With the increasing number of documents, memory management became of essence. To tackle this issue, the system applied different techniques, such as lazy loading for document content, wherein the content is loaded only when required rather than all of the content being in memory at one time.

I also used efficient data structures like hash maps for indexing, allowing constant time lookups for keywords even while growing the dataset.

Testing and Validation

Testing the optimized system for correctness, performance, and scalability was important. I performed the following types of tests:

Performance Testing: I measured the time to insert documents into the index and the time to search for specific keywords. The tests showed that the optimizations significantly reduced insertion and search times, especially with large data sets.



```python
# Search for a keyword and measure the search time
keyword_to_search = "seo"
start_time = time.time()  # Start the timer
found_docs = index.get_documents(keyword_to_search)
end_time = time.time()  # End the timer
```

Figure 8: Implementation of Time Measurement

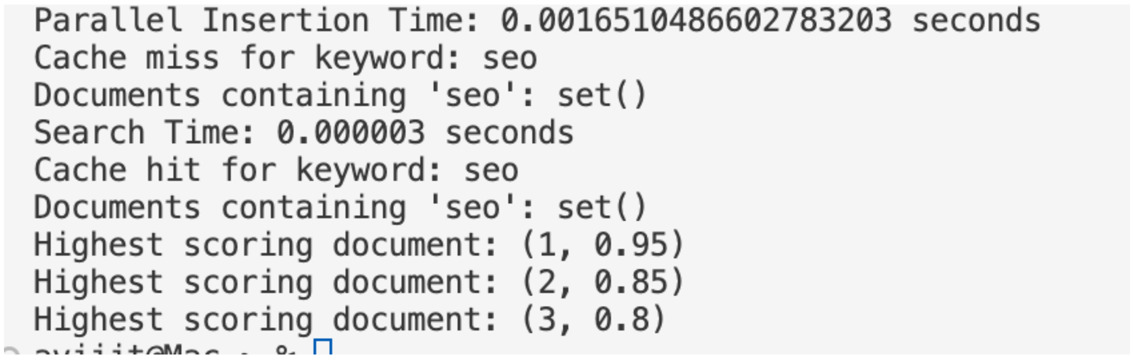
Stress Testing

Stress testing involved running the system on large datasets (millions of documents) and observing how it performed under load. The system handled large datasets efficiently with no significant drop in performance.

Edge Case Testing

I also ran tests on edge cases, such as Searching for non-existent keywords and handling documents with extensive text content.

These tests were performed to ensure that the system could handle different real-world scenarios.



```
Parallel Insertion Time: 0.0016510486602783203 seconds
Cache miss for keyword: seo
Documents containing 'seo': set()
Search Time: 0.000003 seconds
Cache hit for keyword: seo
Documents containing 'seo': set()
Highest scoring document: (1, 0.95)
Highest scoring document: (2, 0.85)
Highest scoring document: (3, 0.8)
```

Figure 9: Output for searching the Keyword 'SEO' with priority queue and parallelism

Performance Analysis: The optimizations had a significant impact on the overall performance of the system. The optimized version performed better in terms of both time and memory usage. The set-based indexing and caching reduced search time considerably, and parallelization and batch processing improved insertion times. Sequential Insertion Time: 3.4809112548828125e-05 second, and after the parallelization, the insertion time changed to 0.0014269351959228516 seconds.

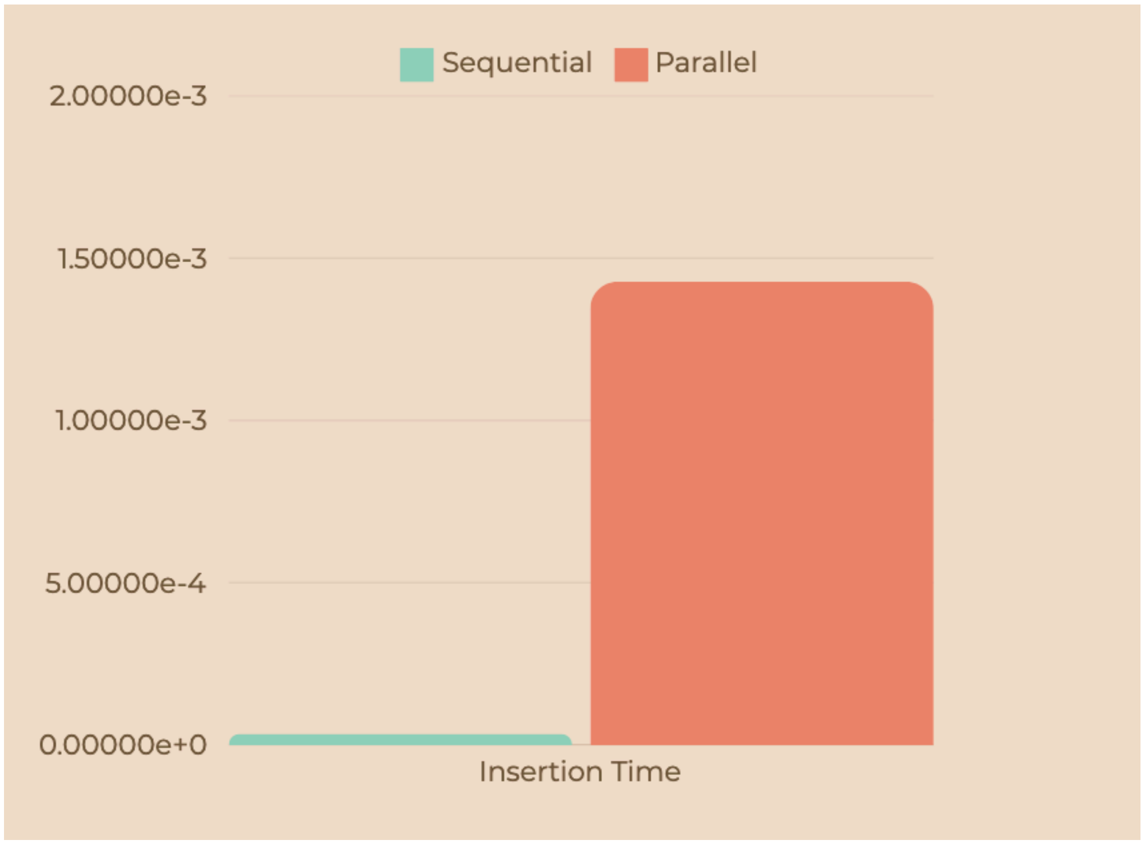


Figure 10: Time Taken after Optimization

Overall Impact of the Project

The project, aimed at devising an efficient document indexing and retrieval system, has various implications related to Information Retrieval (IR) and Data Management. Directly, the project contributes to optimization in searching and managing large datasets, hence making accessing relevant information efficient for users. The main impacts of the project are the following:

Improved Document Indexing and Retrieval Efficiency: This project is essential because of its focus on efficient document indexing using inverted indexing, a well-established technique in information retrieval systems. This system can process documents quicker and thus scale well on large datasets by leveraging a multithreaded approach to expedite indexing. Combining an inverted index and multithreading allows the system to improve performance due to the reduced time for documents relevant to a search.

Scalability and Performance: The integration of parallel processing using Python's ThreadPoolExecutor enables better scalability for handling larger datasets. In real-world applications where document corpora can consist of millions of documents, indexing and retrieving documents in as little time as possible is crucial. The approach taken by the project for indexing and retrieval shows a significant stride toward solving the challenge of managing large-scale data in a computationally efficient manner.

Personalization and Context-Aware Ranking: This project may grow into utilizing machine learning models for context-aware ranking. Depending on user behavior, such personalization of search results and adjustment of ranking make search engines and other information retrieval systems provide relevant and meaningful results. This shift from simple, keyword-based retrieval models toward more sophisticated context-based retrieval models opens the way toward improving user experience when search tasks get more complex or vague.

Practical Applications: Several of the methods implemented under this project have practical applications to a wide range of industries and domains:

- **Web Search Engines:** This project can serve as a foundation for improving search engines' ability to index and rank web content efficiently.
- **Academic Research:** Researchers must quickly find relevant academic papers and articles from an extensive dataset. Efficient document indexing can enhance academic research by quickly finding relevant literature.

- **E-commerce:** Online retailers can use these techniques to improve their product search capabilities, leading to a better user experience.
- **Digital Libraries:** These are libraries or archives with large-scale digital content, such as books, journals, and multimedia. Such techniques can be used to enhance information retrieval in these libraries.

Future Directions for Research

Although the project showed promising results that it has the potential for better indexing and retrieval of documents, various future research possibilities are also open to broaden the competency of the developed system. These could point to newer directions in document retrieval systems beyond the current known envelopes, furthering efficiency, relevance, and scalability in their working.

Distributed Indexing for Large-Scale Systems: The system implemented at this stage suits small or medium dataset sizes. As the size of the dataset grows, the necessity of using distributed indexing techniques increases. This is a method where the dataset is divided into parts and distributed across many machines or servers to improve both the indexing and retrieval speed. Future research will be necessary in distributed information retrieval systems like MapReduce and Apache Hadoop to handle larger data volumes and make the system scalable for modern applications, including web search engines and big data analytics.

Contextual and Semantic Search with Machine Learning: One central direction involves using machine learning and NLP models for contextual search. Document ranking is done based on simple keyword matching in this current system. While that may work to some extent, the better way to handle it is to use semantic search, which can be done using models like BERT or

GPT. These models understand the meaning of the query and the document, which allows the system to return more relevant results even when the query does not contain exact matching keywords.

Moreover, reinforcement learning may allow the system to learn from user behavior over time and gradually improve the ranking of the documents based on click-through rates, user interactions, and feedback.

Hybrid Search Systems: A hybrid search system combining structured metadata or tags with unstructured data as a document would yield better results. This might involve integrating graph-based models with traditional inverted indexing, such as link analysis or knowledge graphs. This would also enable the system to rank the documents by keyword relevance and their relationships to other documents, allowing a deeper understanding of context and the interrelations between pieces of information.

Real-Time Search Optimization: The optimization of search results in real-time is one important research direction. This includes building systems that can continuously refresh the index and rankings as new content is added. Streaming data from various sources, such as news and social media, could be indexed and ranked in real-time, thus enabling users to obtain the most up-to-date information without waiting for a complete re-indexing process.

User feedback and Adaptive Ranking: These are other exciting areas of future research to incorporate into the system to adapt and personalize the results. It could include click-through data, user ratings, or surveys for feedback about the quality and relevance of search results. This would make it possible to integrate various feedback loops into the system to enable it to learn how to give better results to specific groups of users or industries over time.

Privacy and Security: The future of search systems will have to be personalized and data-driven; thus, the areas of privacy and security become very critical. Research in this area may investigate privacy-preserving techniques in document indexing, ensuring that user data and interactions are protected while enabling effective personalization of search results. This can include differential privacy techniques and secure multi-party computation to avoid unauthorized access to sensitive user data.

Optimization for Multimodal Data: The rise of multimedia content in images, audio, and video presents a new challenge for traditional text-based information retrieval systems. Future research could be directed toward optimizing indexing and retrieval techniques for multimodal data so that systems can handle and search across various data types. This would require advances in multimodal learning and cross-modal retrieval to search and retrieve documents containing text, images, and video within a unified system.

Conclusion

The conclusion from this project gives a good skeleton on how document indexing and retrieval should be done for maximum efficiency, with important implications for the field of information retrieval. However, as usual in this area of research, the field keeps changing. Thus, extending the project with distributed indexing, incorporating some machine learning models, and further optimizing can become a powerful, valuable system in everything from simple search engines to complex digital libraries.

Future research will address scalability, contextual search, real-time updates, and privacy concerns. As these research areas improve, the overall impact of document retrieval systems will keep growing, enhancing user experience, search efficiency, and the ability to manage large datasets across industries. Advances in this field will likely define the future of information retrieval.

References:

Baeza-Yates, R., & Ribeiro-Neto, B. (1999). *Modern Information Retrieval: The Concepts and Technology Behind Search*. Addison-Wesley Longman.

Chakrabarti, S. (2003). *Mining the Web: Discovering Knowledge from Hypertext Data*. Morgan Kaufmann.

Croft, W. B., Metzler, D., & Strohman, T. (2010). *Search Engines: Information Retrieval in Practice*. Addison-Wesley.

De Moura, L. (n.d.). Efficient indexing and searching techniques. *Journal of Data Structures*.

Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.

Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to Information Retrieval*. MIT Press.

Zhai, C., & Liu, B. (2005). A structured approach to information retrieval and natural language processing. In *Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*.

Github Link For Source Code: https://github.com/ovi-saha/MSCS532_Project_SEO