Master of Science in
Artificial Intelligence and Data Engineering

Information Retrieval System
Project Report

G.Bello, F.Frati, C.Liu

December 24, 2023

Contents

1 Introduction

1.1 Project Overview

The project at hand is an information retrieval system designed to efficiently process, index, and query collections of textual data. The core of the system lies in its ability to transform unstructured text into a structured, searchable format, enabling fast and accurate retrieval of information. Utilizing data processing and indexing, the system caters to the ever-growing need for handling vast amounts of digital information.

1.2 Motivation

In today's digital age, the amount of data generated and consumed is growing exponentially. This surge in data, especially in textual form, presents a significant challenge in terms of storage, organization, and retrieval. The motivation behind this project stems from the need to address these challenges. By developing a system that can efficiently process and index large datasets and provide quick and relevant responses to user queries, we aim to bridge the gap between the vast stores of data available and the information needs of users.

1.3 Key Objectives

Efficient Indexing: To develop a robust indexing system capable of handling large volumes of data. This involves creating an inverted index that maps terms to the documents they appear in, enabling quick retrieval. Accurate Query Processing: To implement effective query processing algorithms that can parse and interpret user queries, and return relevant results swiftly and accurately.

Optimization of Storage and Retrieval: To utilize techniques like unary and variable byte encoding for optimizing data storage and retrieval, to reduce the system's storage footprint and to improve performance. Scalability and Performance: To ensure that the system is scalable and can maintain high performance as the size of the data grows.

Comprehensive Testing and Evaluation: To thoroughly test the system across various parameters, including accuracy, efficiency, and speed, ensuring its reliability and effectiveness in real-world scenarios.

2 Indexing

2.1 Overview of Indexing Process

The indexing module is one the two main parts of the information retrieval system, designed to handle large-scale data and convert it into an organized, searchable structure. This process involves several stages, from initial document processing to the creation of a comprehensive inverted index.

2.2 Single-Pass In-Memory Indexing

2.2.1 SPIMI Algorithm (Class: SPIMI)

Algorithm Implementation: The SPIMI algorithm is implemented in the SPIMI class. It is designed to efficiently process large document collections within the constraints of available memory, making it ideal for handling extensive datasets.

Document Processing: This stage involves reading and parsing each document in the collection. The SPIMI class manages the reading of documents, in our case compressed in format .tar.gz, and processes them sequentially.

Inverted Index Construction: As the documents are processed, terms are extracted, tokenized, and added to an in-memory inverted index. For each term, the algorithm checks whether it exists in the current index. If so, the existing posting is updated; otherwise, a new entry is created.

Memory Management: A key feature of the SPIMI implementation is its efficient memory management. When the memory usage approaches a set threshold, in our implementation 80%, the current in-memory index is written to disk, and a new indexing process starts. This approach ensures optimal memory utilization without sacrificing processing speed.

Index Writing: The write2Disk method within the SPIMI class is crucial for persisting the in-memory index to disk. It organizes and stores postings in a structured format, enabling efficient data retrieval.

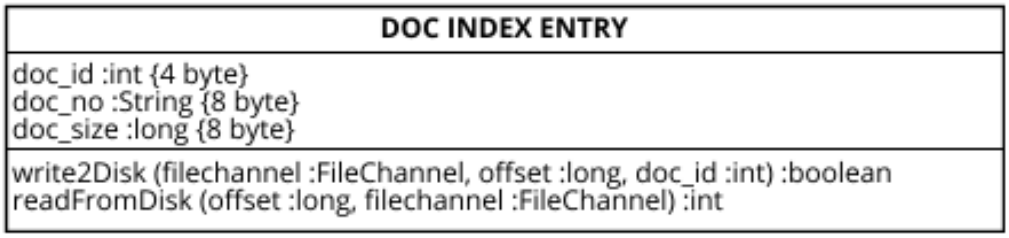
2.2.2 SPIMI Merger (Class: SPIMIMerger)

Merging Partial Indexes: Post-SPIMI, the document collection is represented by multiple partial inverted indexes. The SPIMIMerger class is responsible for merging these into a single, unified index.

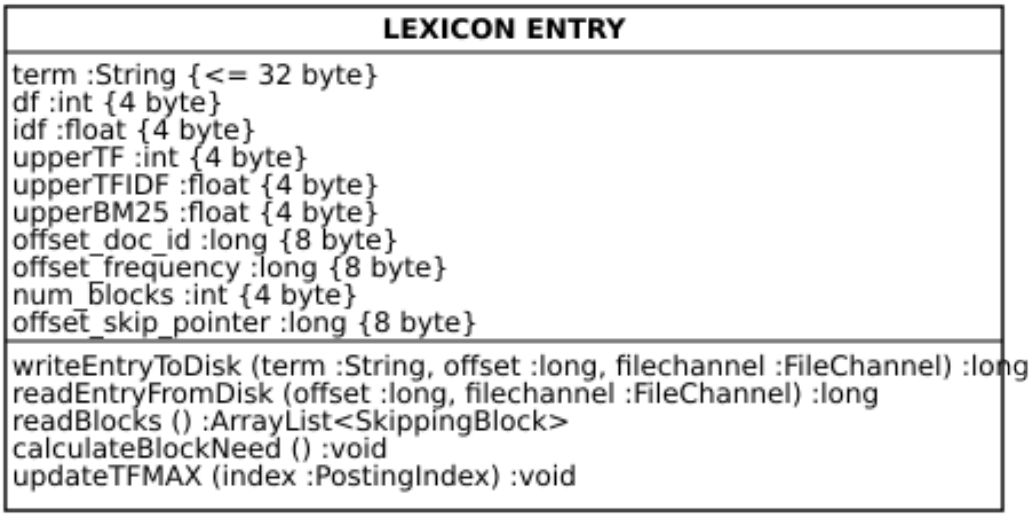
Merger Functionality: The class reads individual index files and merges postings lists for the same terms from different indexes. This process results in a comprehensive inverted index that encompasses the entire document collection.

2.3 Index Structure and Organization

Initial Document Handling (Classes: DocIndex, DocIndexEntry): The foundation of the indexing process begins with the methodical handling and organization of documents. The DocIndex class is instrumental in assigning a unique identifier, known as `doc_id`, to each document in the collection.This identifier is key to tracking and retrieving documents throughout the system. The DocIndexEntry class complements this by storing vital metadata for each document. This metadata includes these information: `doc_id`, `doc_no` and the length of the document(`doc_size`). By systematically managing this data, the system enhances the efficiency and accuracy of subsequent retrieval processes.



Lexicon Construction (Classes: Lexicon, LexiconEntry): The Lexicon is a central component of the indexing structure, serving as a directory for all unique terms found within the document corpus. The Lexicon class creates and maintains this directory, ensuring quick access to term information during query processing. Each term in the lexicon is associated with a LexiconEntry, which contains metadata such as term frequency and pointers to the postings lists in the index.



Posting List Management (Classes: Posting, PostingIndex): Posting lists are at the heart of the inverted index, linking terms to the documents in which they appear, also including: list of skipping blocks, currently active skipping block, currently active posting, two iterator(one for postings and one for skipping blocks), upper bound and inverse document frequency(idf). The Posting class represents individual records in a posting list, encapsulating a document identifier and term frequency within that document. The PostingIndex class manages these lists, organizing them in a way that supports quick searches.

Corpus Statistics Aggregation (Classes: CollectionStatistics): The CollectionStatistics class aggregates important statistical data about the corpus: total number of documents, average document length and total number of terms. This statistical information is used to

refine the system's retrieval strategies, enabling it to adjust scoring and ranking algorithms to the specific dataset.

Preprocessing of Textual Data (Class: Preprocess): Preprocessing is a critical step in transforming raw text into a structured format that is conducive to efficient indexing. The Preprocess class is tasked with this important function. The first stage in preprocessing involves text normalization, where the raw text is cleaned to remove any HTML tags, special characters, and other extraneous elements. This step ensures that only meaningful textual content is processed further. The next stage is tokenization, where the cleaned text is broken down into individual words or terms. This process is vital for analyzing the text at a granular level. After tokenization, the system performs stopword removal. Here, common words that offer little value in search queries, such as "the", "is", and "at", are filtered out. This step significantly reduces the dataset size and focuses on the more meaningful terms within the text. Finally, stemming is applied to the remaining terms. Stemming involves reducing words to their base or root form. For instance, variations like "running", "runs", and "run" are all reduced to the root word "run". This process helps in standardizing terms in the index, ensuring that variations of a word are treated uniformly during search queries.

2.4 Data Encoding and Compression

Unary Encoding (Class: UnaryConverter): The UnaryConverter class is designed to provide an efficient encoding mechanism for data that has a skewed distribution, which is often the case with term frequencies in document collections. In unary encoding, a number is represented by a series of 1's followed by a 0. For example, the number 3 is encoded as "110". This method is particularly effective for encoding smaller numbers, which are common in term frequency data, leading to a more compact representation. Moreover, unary encoding is optimized as it operates at the array level rather than with each individual number, ensuring additional space savings.

Variable Byte Encoding (Class: VariableByteEncoder): The VariableByteEncoder class implements variable byte encoding, a technique that represents integers in a variable number of bytes, making it highly space-efficient for a wide range of integer sizes. This encoding method is especially advantageous for indexing as it allows for the compression of document identifiers and term frequencies without a fixed-size constraint. Each number is broken down into 7-bit chunks, with the most significant bit in each byte used as a continuation marker. This approach ensures that larger numbers occupy more space, while smaller numbers are stored more compactly, leading to overall space optimization.

3 Query Processing

3.1 Overview of Query Processing

Query processing is a critical component of the information retrieval system, responsible for interpreting user queries and retrieving relevant documents from the indexed data. This process involves several key steps, from query parsing to document scoring and ranking.

3.2 Query Parsing and Execution

Advanced Query Interpretation (Class: Processer): The Processer class extends beyond basic parsing to handle various query types, including both simple and complex constructs. The class ensures that queries are broken down, preprocessed (including normalization, tokenization, stopword removal, and stemming), and accurately interpreted, aligning user intent with the indexed data.

Handling Diverse Query Types: Our system accommodating two different query formats: conjunctive (AND) and disjunctive (OR). This capability allows users to refine their search criteria and retrieve results that precisely match their information needs.

3.2.1 Document At A Time (Class: DAAT)

Efficient Document Evaluation: The DAAT class implements the Document At A Time strategy. It involves systematically checking each document for the occurrence and frequency of query terms, thus calculating a relevance score for the document. This approach is especially effective for multi-term queries (both conjunctive and disjunctive) as it allows the system to assess all query terms simultaneously for each document, ensuring comprehensive and accurate scoring.

3.2.2 Dynamic Pruning (Class: MaxScoreDynamicPruning)

Maximizing Query Efficiency: The MaxScoreDynamicPruning class plays a significant role in optimizing query processing. This class employs a dynamic pruning strategy based on the maximum possible score a document can achieve. By estimating this score and comparing it against a relevance threshold, the system can effectively disregard documents unlikely to be relevant, reducing the computational load by limiting the number of documents that need full evaluation without compromising result quality.

3.2.3 Leveraging Search Algorithms for Varied Queries

Conjunctive and Disjunctive Queries: For conjunctive queries, where all terms must be present in a document, the system applies strict matching criteria, ensuring that only documents containing all query terms are retrieved. In contrast, for disjunctive queries, where any of the terms can be present, the system adopts a more inclusive approach, broadening the search scope to retrieve any document containing at least one of the query terms.

3.3 Scoring and Ranking

Determining Document Relevance (Class: Scorer): One of the main challenges in query processing is accurately scoring each document considering the relevance for the user, a task undertaken by the Scorer class. This class employs two scoring models: TF-IDF (Term Frequency-Inverse Document Frequency) and BM25, which assess the significance of query terms within

each document. This relevance scoring is crucial for understanding the degree of match between the document's content and the user's query.

Ranking Mechanism (Class: TopKPriorityQueue): Once documents are scored, the next step is to rank them in order of relevance. The TopKPriorityQueue class is instrumental in this process. This specialized priority queue maintains a limited set of the highest-scoring documents, often referred to as the 'Top K' results. As new documents are scored, they are evaluated against the lowest-scoring document in the queue. If a new document scores is higher, it is included in the queue, replacing the lower-scoring document. This ensures that only the most relevant documents are retained, optimizing memory usage and computational resources.

Handling Different Query Types: Conjunctive Queries: For conjunctive (AND) queries, the system focuses on documents that contain all query terms. The scoring models are tuned to prioritize documents where all terms are present, ensuring the relevance of the results to the query's strict criteria.

Disjunctive Queries: In contrast, disjunctive (OR) queries require a broader approach. The scoring algorithms are adjusted to accommodate documents containing any of the query terms. This flexibility is key to retrieving a comprehensive set of relevant documents for queries with multiple, potentially independent terms.

Final Ranking and Presentation: After the scoring process, the documents in the TopKPriorityQueue are sorted based on their relevance scores, in descending order. This final ranking dictates the order in which documents are presented to the user, with the most relevant documents appearing first. This is really important because the user is generally impatient and often only checks the very first documents returned.

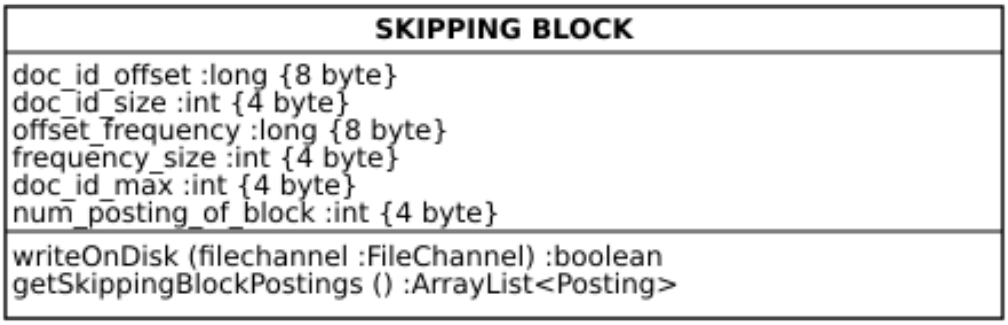
3.4 Optimization Techniques

Efficient memory management is a central task in high-performance information retrieval systems, because rapid access to frequently used data and optimized overall system response are needed. Two techniques implemented in our system are the LFUCache (Least Frequently Used Cache) and the SkippingBlock.

Prioritizing Frequently Accessed Data (Class: LFUCache): The LFUCache class is a strategic approach to in-memory data optimization. It operates on the principle that frequently queried terms or documents should be readily accessible, minimizing latency and accelerating query response times. This is particularly salient in the field of information retrieval, where the query frequency of certain terms can vary. The LFUCache's algorithm allows that the most relevant data remains in the forefront of memory allocation.

Streamlining Index Traversal (Class: SkippingBlock): Conversely, the SkippingBlock class is used to refine the index traversal process during query execution. Its utility is evident when the system confronts lengthy posting lists, a common occurrence for prevalent query terms. By enabling the system to bypass non-critical sections of these lists, SkippingBlock makes exhaustive linear traversal not mandatory, reducing the search duration required to locate pertinent documents.

Synergistic Impact on Query Processing: The synergy between LFU caching and skipping blocks represents a paradigm shift in query processing efficiency. LFU caching's contribution lies in reducing memory access times, while skipping blocks reduce the computational load during the index search.


SKIPPING BLOCK
doc_id_offset :long {8 byte}
doc_id_size :int {4 byte}
offset_frequency :long {8 byte}
frequency_size :int {4 byte}
doc_id_max :int {4 byte}
num_posting_of_block :int {4 byte}
writeOnDisk (filechannel :FileChannel) :boolean
getSkippingBlockPostings () :ArrayList<Posting>

4 Performance Evaluation

4.1 Evaluation Methodology

4.1.1 Overview of Evaluation Approach

To rigorously assess the performance of the information retrieval system, we adopted a methodology grounded in the widely recognized Text Retrieval Conference (TREC) standards. This approach ensures that our evaluation is both comprehensive and comparable to industry benchmarks.

4.1.2 Description of Metrics

The following TREC metrics were employed to evaluate retrieval effectiveness.

Precision at 10 (P@10): This metric provided insights into the precision of the system, indicating the proportion of relevant documents within the top 10 search results.

Recall at 10 (R@10): It helped us understand the system's ability to retrieve all relevant documents, albeit within a small subset of the top 10 documents.

Normalized Discounted Cumulative Gain at 10 (ndcg@10): This metric assessed the ranking quality of the search results, indicating how well the system ordered the documents by relevance.

Mean Average Precision at 10 (map@10): It offered a granular view of the system's precision across the rank positions of the retrieved documents, averaged at the cut-off point of 10 documents.

4.2 Indexing Performance

4.2.1 SPIMI Processing Efficiency

Evaluating the efficiency of SPIMI is critical in understanding how quickly our system can translate raw data into a searchable index. Across different configurations, SPIMI processing times varied, with the quickest processing observed when both compression and text preprocessing were disabled. This suggests that while text preprocessing introduces additional computational steps, its absence can lead to an increased index size, potentially affecting the indexing performance negatively.

Without Compression and Preprocessing: SPIMI completed in 986 seconds, demonstrating swift raw data processing.

With Compression, Without Preprocessing: The processing time increased to 1069 seconds, indicating a trade-off where compression efficiency is offset by the time required to compress the data.

Without Compression, With Preprocessing: The time taken rose to 1583 seconds, highlighting the computational cost of stemming and stopword removal.

Optimal Configuration: With both features enabled, the indexing process was expectedly longer due to the additional preprocessing overhead.

4.2.2 SPIMI Merger Time

The SPIMI Merger consolidates individual index segments into a single comprehensive index. Its performance directly impacts the overall time efficiency of the indexing phase.

With and Without Preprocessing: The merger times were significant, at 4105 seconds with preprocessing enabled and 4086 seconds without, suggesting that the merger operation's

complexity and time consumption are relatively invariant to the text preprocessing but sensitive to the volume of data being processed.

4.3 Storage Efficiency

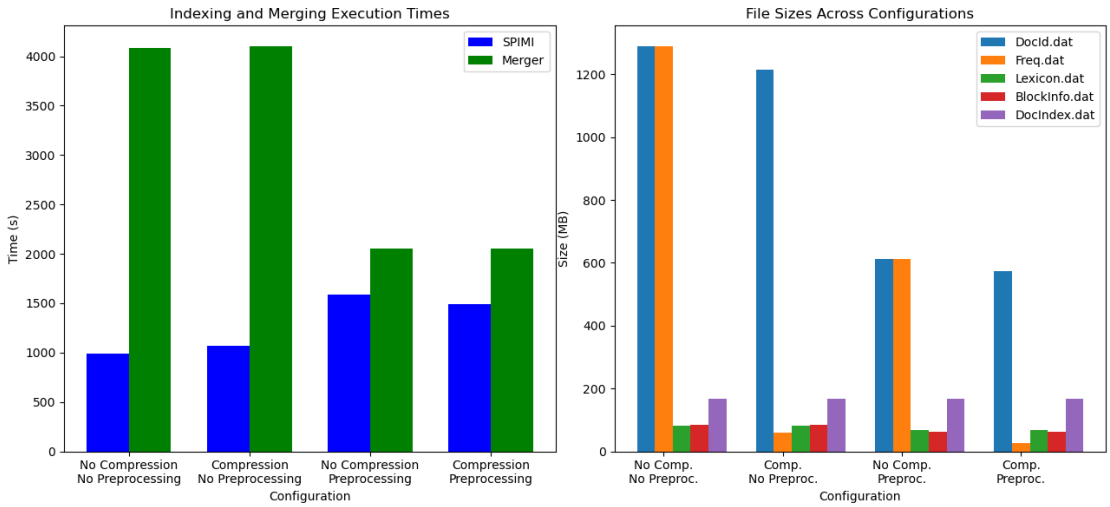
Storage efficiency is a critical aspect of system performance, especially for large-scale information retrieval systems where index size can have a direct impact on both storage costs and retrieval speed. In our system, the use of compression significantly reduced the size of index files, which was expected. However, the absence of text preprocessing led to an increase in the size of index files due to the retention of a larger set of terms.

Without Compression and Preprocessing: Index files such as DocId.dat and Freq.dat reached sizes upwards of 1200 MB, demonstrating the storage demands when handling unprocessed data in its entirety.

With Compression, Without Preprocessing: Even with compression enabled, the index files remained substantially large, albeit reduced compared to their uncompressed counterparts.

Without Compression, With Preprocessing: The inclusion of stemming and stopword removal, while not affecting the size of structural files like Lexicon.dat, BlockInfo.dat, and DocIndex.dat, did result in smaller DocId.dat and Freq.dat files, indicating the reduction of redundant and non-essential data.

Both Compression and Preprocessing Enabled: This configuration offered the most storage-efficient solution, balancing the reduced file sizes with the benefits of preprocessing. Our evaluation highlighted that both compression and text preprocessing play significant roles in optimizing the storage space required for index files. Compression directly reduces file size, whereas text preprocessing decreases the volume of data to be compressed. The effects were most pronounced in Freq.dat files, where the numbers could be efficiently encoded due to repetitive structures that lend themselves well to compression algorithms.



4.4 Query Processing Efficiency

The agility of an information retrieval system is often evaluated based on its query processing efficiency — the speed and effectiveness with which it can parse, search, and retrieve relevant

documents in response to a user's query. We have rigorously tested our system across different configurations to measure this efficiency, particularly focusing on the time taken to process queries and the impact of caching on enhancing this metric.

Without Caching: Query processing times without the benefit of caching reflect the system's inherent performance. The TFIDF DAAT approach, for example, had a maximum processing time of 1087 milliseconds, highlighting the potential latency involved in retrieving query results when solely relying on the system's raw processing capability.

With Caching: The introduction of caching markedly improved processing times. For instance, the mean query response time for TFIDF DAAT was reduced to around 53 milliseconds, demonstrating the significant role of caching in optimizing query processing efficiency.

DAAT Performance: The performance of the DAAT approach without caching showed considerable variance. The mean response times for both BM25 and TFIDF scoring models provided a baseline measure of the system's efficiency.

Dynamic Pruning Performance: In contrast, the Dynamic Pruning approach illustrated its effectiveness in reducing computational overhead. This was evidenced by the decrease in both mean and maximum query times, especially noticeable when caching was enabled.

The juxtaposition of Document-at-a-Time (DAAT) and Dynamic Pruning (DP) approaches offered additional insights. While the DAAT method ensures thorough document evaluation, Dynamic Pruning streamlines the process by strategically eliminating candidates unlikely to be relevant, thus expediting the retrieval process.

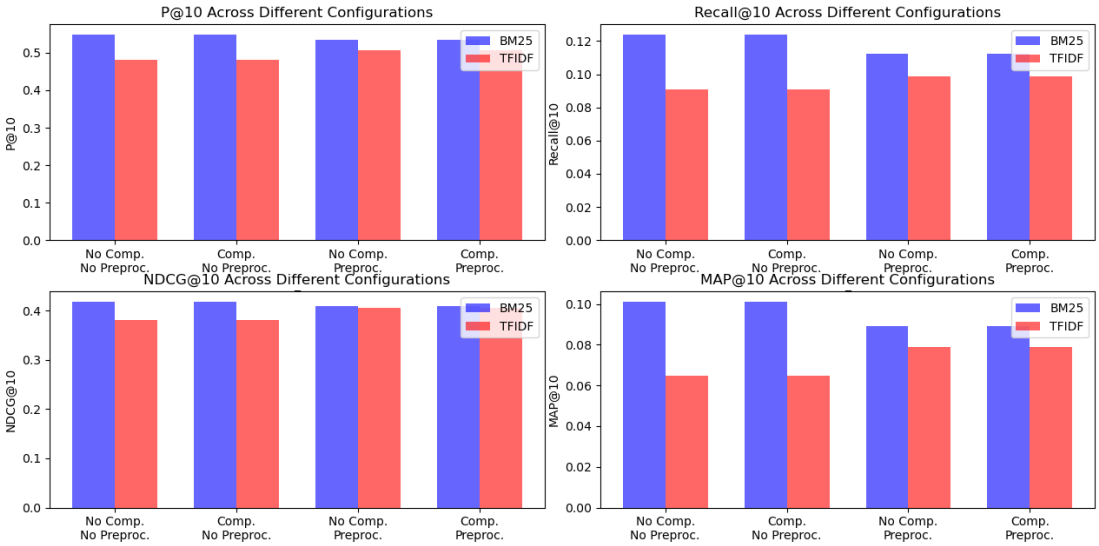
Caching emerged as a significant performance enhancer across all configurations. By temporarily storing frequently accessed data in memory, caching lessens the dependency on slower disk reads. This optimization was particularly evident in the reduced query processing times, confirming caching's pivotal role in boosting the system's overall efficiency.

These findings are visually represented in the accompanying graph, which illustrates the comparative query execution times with and without caching for both TFIDF and BM25 methods, using DAAT and Dynamic Pruning. The error bars in the graph indicate the standard deviation, providing an indication of the variability in the query times and further emphasizing the caching's impact on performance consistency.

4.5 Retrieval Effectiveness

DAAT vs. Dynamic Pruning: Both DAAT and Dynamic Pruning, when paired with BM25 and TFIDF scoring models, demonstrated similar patterns in performance across configurations. This suggests that the choice of retrieval model did not significantly impact the effectiveness metrics in the context of the configurations tested.

Impact of Compression and Preprocessing: Disabling compression and preprocessing yielded the highest P@10 scores in BM25 and TFIDF models, suggesting that the system is capable of retrieving highly relevant documents even with raw, uncompressed data. Enabling compression, regardless of preprocessing, did not significantly alter the P@10 scores, pointing to the effectiveness of the system's compression algorithm in maintaining retrieval quality. The activation of stemming and stopword removal, whether with or without compression, resulted in a slight decrease in P@10. This could be due to the elimination of some terms that were relevant to the retrieval of the top documents. The comparison between configurations revealed that while enabling stemming and stopword removal typically enhances retrieval effectiveness by focusing on more meaningful content, in our system, it slightly reduced the precision in the top 10 results. This unexpected result warrants further investigation into the preprocessing logic and its interaction with the scoring algorithms.


P@10 Across Different Configurations
Recall@10 Across Different Configurations
NDCG@10 Across Different Configurations
MAP@10 Across Different Configurations
BM25
TFIDF
P@10
Recall@10
NDCG@10
MAP@10
No Comp. No Preproc.
Comp. No Preproc.
No Comp. Preproc.
Comp. Preproc.

5 Conclusion

A comparative analysis serves to distill the nuanced effects of different system configurations on retrieval performance. By juxtaposing the four distinct setups—varying the application of compression and text preprocessing—we gain insight into the interplay between these features and their collective impact on the system's ability to retrieve relevant documents.

Without Compression and Preprocessing: This configuration represented the system's baseline capabilities, handling the most comprehensive data set. Surprisingly, it provided high P@10 values, indicating that the system's retrieval algorithms are adept at sifting through unfiltered data to find relevant information.

With Compression, Without Preprocessing: Implementing compression alone showed a negligible impact on the precision of the top results. The retrieval effectiveness remained consistent, underscoring the efficiency of the compression algorithm used.

Without Compression, With Preprocessing: Activating text preprocessing without compression offered a slight dip in retrieval precision. This suggests that the elimination of certain terms via stemming and stopword removal may have removed some relevant documents from the top results.

Both Compression and Preprocessing Enabled: The optimal configuration, in theory, presented a similar slight decrease in P@10, calling into question the interaction between preprocessing and the scoring models.

Stemming/Stopword Removal: Contrary to expected outcomes, enabling stemming and stopword removal did not consistently improve retrieval effectiveness across the board. This indicates that the process may be overly aggressive, potentially filtering out relevant terms in certain contexts.

Compression Stability: The compression feature's stability across different configurations was evident, as it did not adversely affect retrieval precision. This stability is beneficial, especially considering the reduced storage footprint.

Scoring Model Consistency: Both BM25 and TFIDF scoring models exhibited consistent performance regardless of the underlying data processing, suggesting a level of robustness in the system's scoring algorithms.

The findings from this comparative analysis suggest that the system's retrieval algorithms are capable of maintaining a high degree of precision in the top search results, even when dealing with raw data. However, the slight decline in retrieval effectiveness with the introduction of preprocessing highlights the need for a more nuanced approach to text preprocessing, potentially involving a less aggressive filter or more intelligent term weighting.

Table 1: Query Times(ms) for Different Configurations

Configuration	Method	Cache	Mean	Std Dev
Without Compression and Preprocessing	TFIDF DAAT	Without Cache	52.56	78.7921
	TFIDF DAAT	With Cache	28.66	58.7566
	BM25 DAAT	Without Cache	1252.935	5396.1478
	BM25 DAAT	With Cache	1090.785	4728.6081
	TFIDF DP	Without Cache	28.195	55.3017
	TFIDF DP	With Cache	23.935	39.8946
	BM25 DP	Without Cache	819.465	1252.9742
	BM25 DP	With Cache	720.405	1107.8738
Both Compression and Preprocessing Enabled	TFIDF DAAT	Without Cache	16.83	24.5028
	TFIDF DAAT	With Cache	9.625	13.0289
	BM25 DAAT	Without Cache	275.98	359.3345
	BM25 DAAT	With Cache	205.165	265.4148
	TFIDF DP	Without Cache	12.145	17.9330
	TFIDF DP	With Cache	9.71	14.2942
	BM25 DP	Without Cache	239.7	314.1416
	BM25 DP	With Cache	217.475	287.3795
Without Compression, With Preprocessing	TFIDF DAAT	Without Cache	11.995	21.6699
	TFIDF DAAT	With Cache	5.105	9.2743
	BM25 DAAT	Without Cache	262.955	344.0654
	BM25 DAAT	With Cache	206.685	260.6006
	TFIDF DP	Without Cache	6.9	17.3398
	TFIDF DP	With Cache	7.125	19.2984
	BM25 DP	Without Cache	241.255	344.4843
	BM25 DP	With Cache	211.64	273.3636
With Compression, Without Preprocessing	TFIDF DAAT	Without Cache	62.525	99.0562
	TFIDF DAAT	With Cache	53.77	81.1760
	BM25 DAAT	Without Cache	1189.825	5435.5724
	BM25 DAAT	With Cache	1074.14	4789.7906
	TFIDF DP	Without Cache	57.455	75.4365
	TFIDF DP	With Cache	47.58	48.8474
	BM25 DP	Without Cache	831.06	1243.3837
	BM25 DP	With Cache	812.0	1353.8189

6 Limitations and Further Improvements

The following improvements are proposed for future development:

- **Optimizing Preprocessing Steps**: Streamlining the preprocessing algorithms to reduce their complexity and execution time. This could involve implementing more efficient text cleaning, tokenization, stopword removal, and stemming techniques or utilizing parallel processing to accelerate these tasks.
- **Refining Lexicon Parameters**: Adjusting the Lexicon class to streamline its operation, potentially reducing memory usage and improving term lookup times. This might include optimizing the data structures used for storing terms and their associated information or introducing more efficient indexing strategies.
- **Segmenting DocIndex**: Splitting the DocIndex class into smaller sub-indices or partitions to facilitate faster document retrieval. The segmentation strategy should ensure that the system can still effectively manage and query the index partitions.
- **Optimizing SkippingBlock Instances**: Modifying the SkippingBlock class to use fewer instances or more strategically placed instances. This optimization aims to save memory while maintaining or even enhancing the efficiency of search capabilities, especially in navigating long posting lists during query processing.

These proposed improvements and optimizations aim to fine-tune the system's performance, finding a balance between reducing complexity, enhancing memory efficiency, and maintaining or increasing processing speed. Each proposed change will need to be thoroughly tested and evaluated in practical scenarios to ensure that they indeed contribute positively to the system's overall functionality and user experience.