Assignment 1 - Indexing and retrieval

Prepared by

Kushagra Gupta

(2025909001)

Under the guidance of

Prof. Anil Nelakanti

Submitted in
partial fulfillment of the requirements
for the course of

CS4.406 Information Retrieval and Extraction

INTERNATIONAL INSTITUTE OF
INFORMATION TECHNOLOGY

HYDERABAD

(October 2025)

1. Introduction:

This report details the implementation and evaluation of an information retrieval system as part of the [Your Course Name] course assignment. The primary goal was to gain a practical understanding of search index internals by building and comparing different indexing and retrieval strategies.

The project involved two main parts:

- **Utilizing Elasticsearch:** An industry-standard search engine was used to index datasets and establish a performance baseline. This involved data preprocessing, indexing via the Python client, and evaluating query performance using standard metrics.
- **Building** SelfIndex**:** A custom search index was built from scratch in Python, starting with a simple Boolean index with positional information (x=1, y=1) and incrementally adding features like ranking (word counts x=2, TF-IDF x=3), different datastore backends (y=2 - SQLite, Redis), compression techniques (z=1, z=2), index optimization (i=1 - skipping), and alternative query processing strategies (q=D - Document-at-a-time).

All experiments were conducted using two primary datasets:

A collection of news articles sourced from webz.io (obtained via local folders).

Wikipedia articles (English split 20231101.en) obtained via the Hugging Face datasets library.

Performance was evaluated based on the assignment criteria:

Latency (A): p95 and p99 query response times.

Throughput (B): Queries per second.

Memory (C): Disk usage and/or in-memory footprint (RSS).

Functional Metrics (D): Precision and Recall @ 5 against a generated gold standard.

This report is structured as follows: Part 1 describes the Elasticsearch implementation and baseline evaluation. Part 2 details the iterative development and comparative evaluation of the various SelfIndex configurations. Finally, the Conclusion summarizes the key findings and trade-offs observed.

2. Part 1: Elasticsearch Baseline (ESIndex-v1.0):

This section describes the process of indexing the news dataset into Elasticsearch to establish a performance and relevance baseline against which the custom SelfIndex implementations will be compared.

Data Loading & Preprocessing:

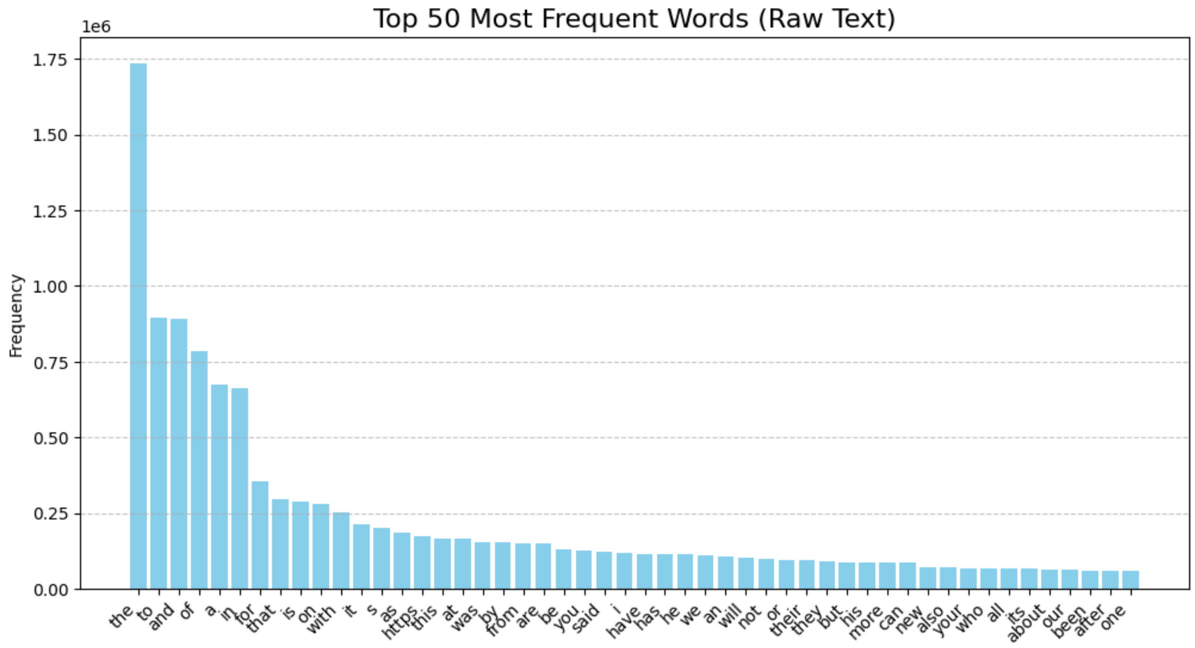
The news data, consisting of numerous JSON files organized into subdirectories, was loaded recursively using a custom Python function (load_all_news_data). This function iterated through all subfolders, read each JSON file , and extracted relevant fields like id, title, and text into a pandas DataFrame.

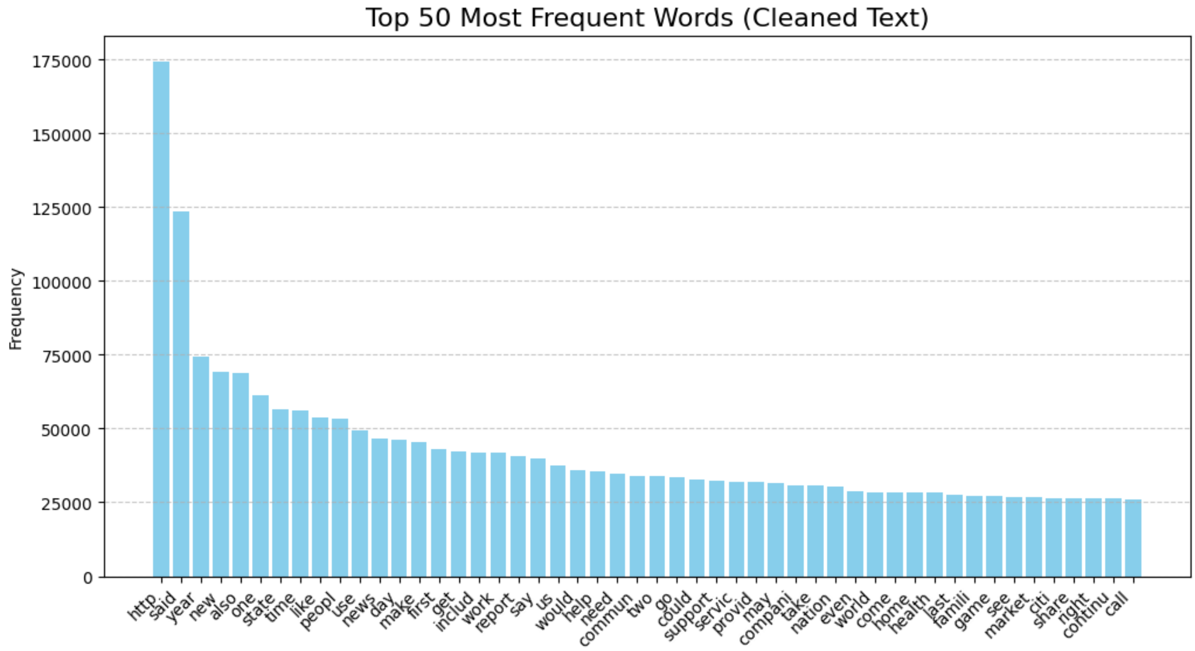
Given the presence of multiple languages, the langdetect library was used to identify the language of each article's text, and the DataFrame was filtered to retain only English articles (filter_by_language).

A standard text preprocessing pipeline (preprocess_text) was applied to the combined title and text fields. This pipeline involved:

1. Lowercasing.
2. Tokenization using nltk.word_tokenize.
3. Removal of punctuation and non-alphabetic tokens (using isalpha()).
4. Removal of standard English stopwords (from nltk.corpus).
5. Stemming using nltk.PorterStemmer.

The impact of this preprocessing is illustrated below, showing the top 50 most frequent words before and after applying the pipeline. As expected, raw text frequencies are dominated by common stopwords, while the cleaned text reveals more meaningful terms.





Indexing:

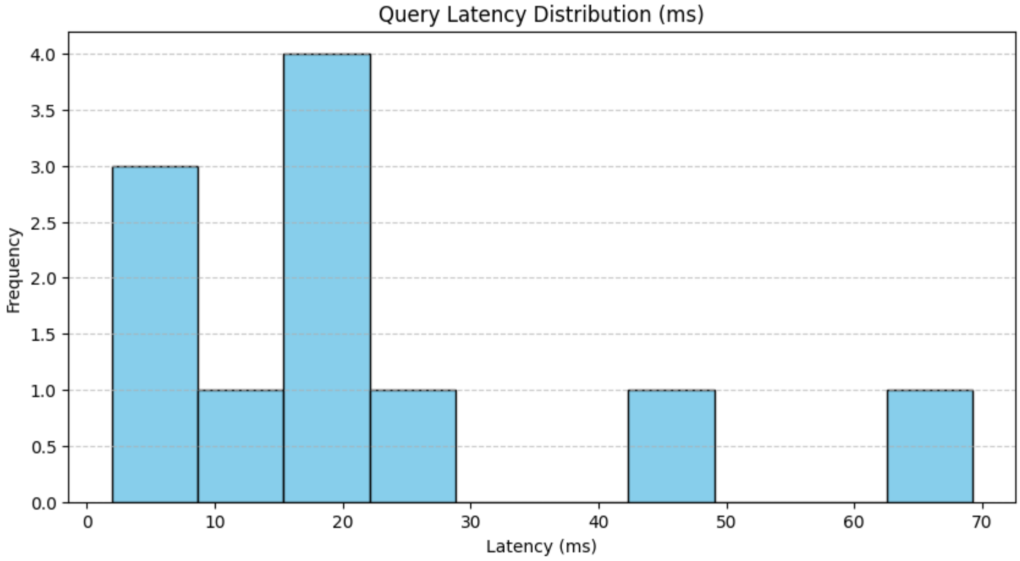
The preprocessed English news articles were indexed into an Elasticsearch (version 8.15.0) index named esindex-v1-0. A specific mapping was defined: title, text, and clean_text were mapped as text fields to enable full-text search, while fields like id, author, categories, and sentiment were mapped as keyword for exact matching and filtering. The published field was mapped as date. Indexing was performed efficiently using the elasticsearch.helpers.bulk API.

The final index contained 61,497 documents.

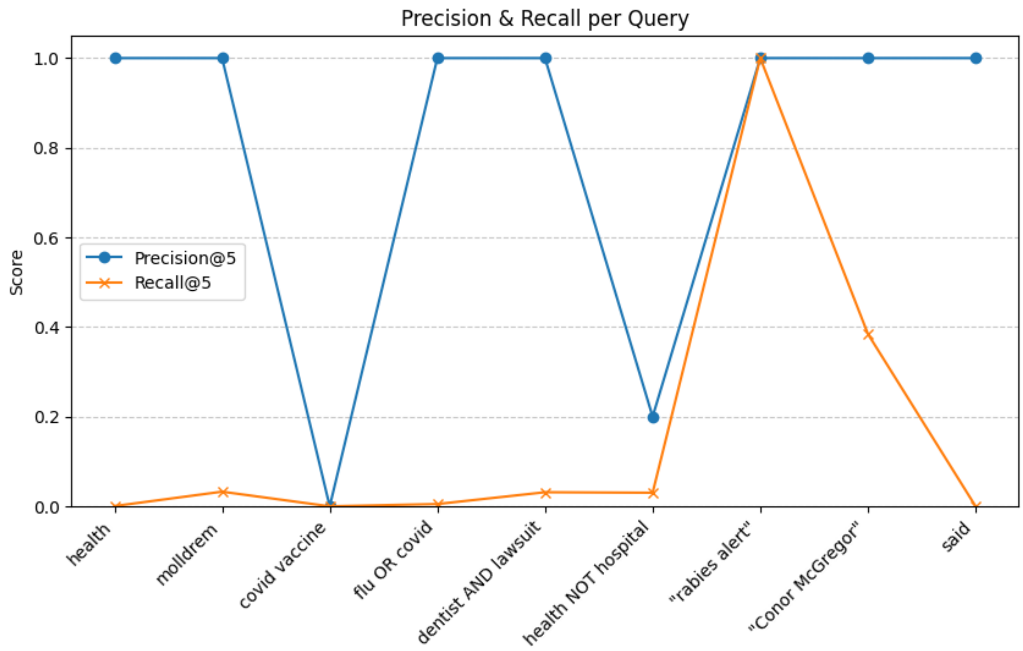
Evaluation:

The esindex-v1-0 index was evaluated using the final corrected evaluate_elasticsearch function and the generated gold_standard. The key performance metrics were:

- **Metric A (Latency):** p95 = 44.16 ms, p99 = 59.70 ms.
- **Metric B (Throughput):** 237.91 queries/sec.
- **Metric C (Memory):** 433.28 MB (JVM Heap Used).



- **Metric D (Precision & Recall @ 5):**
  - Precision was generally high (1.00 for most queries), indicating the top 5 results were relevant according to the boolean gold standard.
  - Recall was low (0.00-0.03) for broad queries, as expected for R@5 when many documents are relevant.
  - Phrase queries ("rabies alert", "Conor McGregor") achieved perfect precision and high recall after fixing the query generation to use match_phrase with preprocessed terms.
  - Some queries (covid vaccine, health NOT hospital) showed low precision, likely due to differences between Elasticsearch's BM25 ranking and the gold standard's boolean logic.



These results provide a solid baseline for comparing the performance and relevance of the custom SelfIndex implementations detailed in the next section.

3. Part 2: SelfIndex Implementation & Evaluation

This section details the development and evaluation of the custom SelfIndex, built incrementally according to the assignment specifications (xyziq).

3.1 Base Implementation (SelfIndex - x=1, y=1, q=T, i=0, z=0):

The foundational version of SelfIndex was implemented, inheriting from the provided IndexBase abstract class.

- **Core Structure:** This version utilizes a standard Python dictionary as the in-memory inverted index.
- **Information Indexed (x=1):** It stores positional postings lists, mapping each term to a dictionary of document IDs, where each document ID maps to a list of integer positions where the term occurs (e.g., {'term': {'doc_id': [pos1, pos2]}}). This fulfills the x=1 requirement.
- **Datastore (y=1):** Persistence is achieved using Python's built-in pickle module, saving the entire index data structure (including postings, document metadata, etc.) to a single file on the local disk. This fulfills the y=1 requirement.
- **Query Processing (q=T):** A Term-at-a-Time boolean query engine was implemented. It uses the Shunting-yard algorithm (_shunting_yard) to parse infix queries (including parentheses and AND/OR/NOT operators respecting precedence) into Reverse Polish Notation (RPN). An RPN evaluator (_evaluate_rpn) then processes these queries using set operations (intersection, union, difference) on document ID sets retrieved from the index. Phrase queries ("quick brown") are handled by retrieving positional postings and checking for adjacency (_eval_term_to_set).

Evaluation (Full Dataset):

This base SelfIndex was evaluated on the full English news dataset (61,497 documents).

- **Metric A (Latency):** p95 = 11.88 ms, p99 = 11.98 ms.
- **Metric B (Throughput):** 524.44 queries/sec.
- **Metric C (Memory):**
  - **Disk (Pickle File):** 253.67 MB.

    - **In-Memory (RSS Estimate):** 842.45 MB.

- **Metric D (Precision & Recall @ 5):** Precision was generally perfect (1.00), aligning well with the boolean gold standard. Recall remained low for broad queries, as expected. Phrase queries were handled correctly.

Comparison to Elasticsearch:

Compared to the Elasticsearch baseline, this initial SelfIndex demonstrated significantly faster query latency (p95: ~12 ms vs. ES ~44 ms) and higher throughput (~524 q/s vs. ES ~238 q/s). This speed advantage is attributed to the in-memory nature of the Python dictionary lookups and set operations versus Elasticsearch's server overhead. However, this came at the cost of higher application memory usage (RSS: ~842 MB) compared to Elasticsearch's managed JVM heap (~433 MB), as the entire index is loaded into the Python process's memory. The disk footprint was considerably smaller than the space Elasticsearch might use internally.

3.2 Information Indexed Comparison (x=n):

This section compares SelfIndex variants based on the type of information stored in the postings list, corresponding to x=1, x=2, and x=3 in the assignment specification.

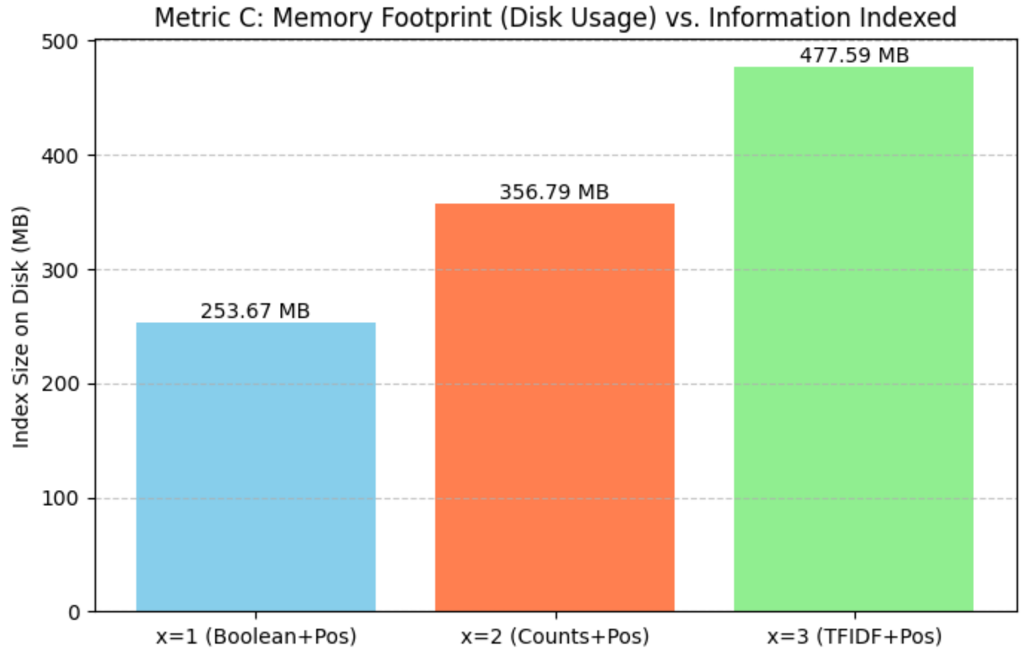
x=1 (Boolean + Positions): The base implementation described in 3.1, storing only term positions: {'term': {'doc_id': [pos1, pos2]}}.

x=2 (Word Counts + Positions): Implemented as SelfIndexRanked. This version modified the index structure to include the term frequency (count) alongside positions: {'term': {'doc_id': {'count': N, 'pos': [...]}}}. Querying was updated to use these counts for basic ranking.

x=3 (TF-IDF + Positions): Implemented as SelfIndexTFIDF. This built upon x=2, adding the calculation and storage of Document Frequency (DF) for each term and the total document count (N) during index creation. Querying was updated to calculate and rank by TF-IDF scores `(TF×IDF, where IDF=log(N/df))`.

Evaluation (Metric C - Disk Usage):

The primary metric for comparing these variants was the disk space required by the persisted index file (.pkl for y=1). The plot below shows the disk usage for indexes built on the full dataset for each x variant.



Analysis:

As shown in the plot, adding word counts (x=2) resulted in a noticeable increase in disk size compared to storing only positions (x=1). This is expected as additional integer counts are stored for every term occurrence in every document. Adding Document Frequency information for TF-IDF (x=3) resulted in only a marginal increase in disk size compared to x=2, as the DF dictionary is relatively small compared to the size of the full postings lists.

3.3 Datastore Comparison (y=n):

This section evaluates the impact of different backend datastores on index performance, specifically comparing the custom file-based approach (y=1) against two off-the-shelf databases (y=2), using the TF-IDF index (x=3) as the base.

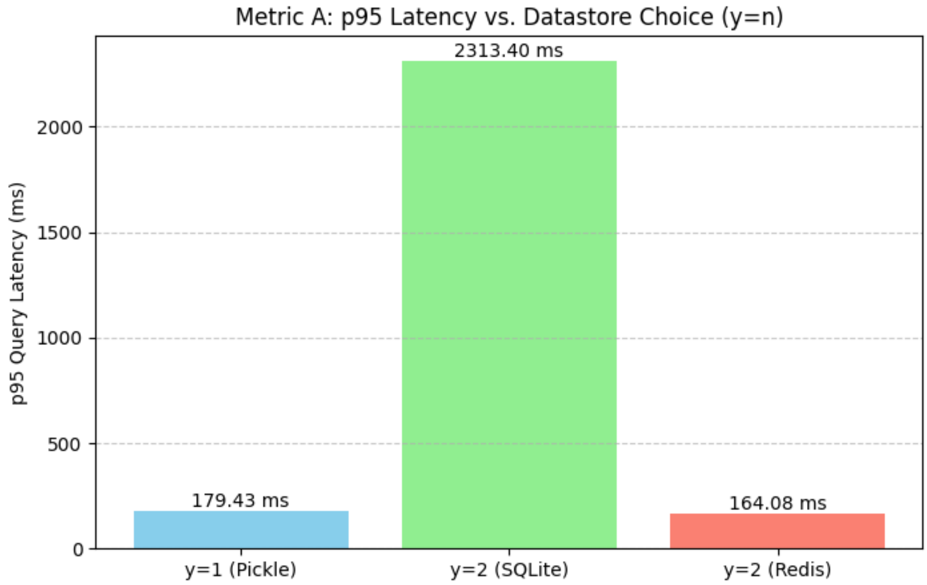
- y=1 (Pickle): The SelfIndexTFIDF implementation stored the entire index structure (inverted index, document frequencies, etc.) in a single Python pickle file on disk. This requires loading the whole index into application memory for querying.
- y=2 (SQLite - DB1): The SelfIndexSQLite implementation stored the index

across multiple tables (terms, documents, postings, metadata) in a serverless SQLite database file. Queries interacted with the database using SQL, avoiding loading the full index into application RAM.

- y=2 (Redis - DB2): The SelfIndexRedis implementation stored the index in an in-memory Redis server (running via Docker). It utilized Redis Hashes for postings and document frequencies, Sets for document IDs, and Strings for metadata. Queries interacted with the Redis server over the network.

Evaluation (Metric A - Latency):

The primary comparison metric was p95 query latency, evaluated on the full dataset.



Analysis:

The plot clearly shows significant performance differences:

- **SQLite (y=2, DB1)** was by far the slowest (p95: ~2313 ms). This is attributed to the high cost of disk I/O required for SQL lookups during query processing.
- **Pickle (y=1)** offered good performance (p95: ~179 ms) once the index was loaded into RAM, benefiting from fast in-memory dictionary access. However, it incurred high application memory (RSS) usage.
- **Redis (y=2, DB2)** provided the fastest query latency (p95: ~164 ms). Being an in-memory datastore, it avoids disk I/O bottlenecks , and its optimized C implementation potentially outperformed the pure Python dictionary lookups

of the Pickle approach. Like SQLite, it kept application memory usage low as the index resided in the separate Redis server process.

Pros & Cons Summary:

Datastore	Pros	Cons
Pickle	Simplest implementation; Very fast queries (once loaded)	High application RAM usage; Slow initial load for large indexes; Basic persistence
SQLite	Simple setup (serverless); Low application RAM usage; ACID persistence	Very slow queries due to disk I/O; Indexing potentially slow; Not ideal for high concurrency
Redis	Very fast queries (in-memory); Low application RAM usage; Shareable; Good data structures	Requires separate server; High server RAM usage; Network overhead; Requires learning Redis API

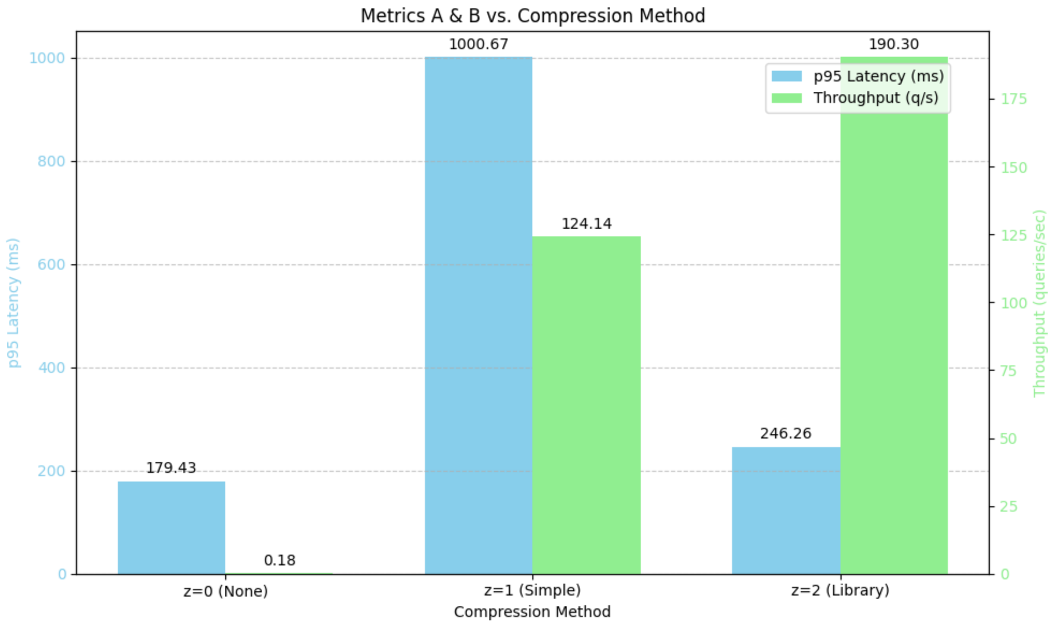
3.4 Compression Comparison (z=n):

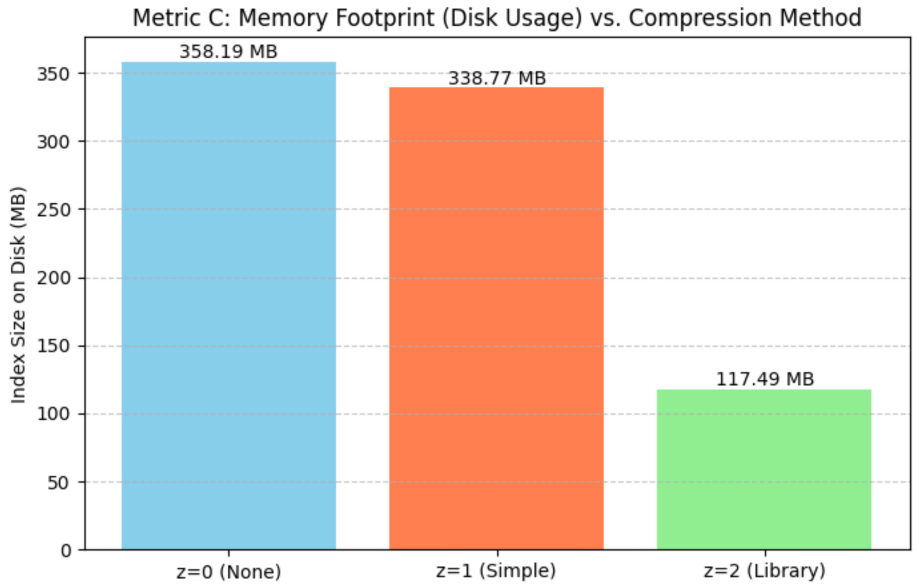
This section analyzes the impact of applying compression techniques to the TF-IDF index (x=3, y=1) to reduce its disk footprint. Two methods were implemented:

- z=1 (Simple Compression): Implemented in SelfIndexCompressedSimple. This method applied Delta Encoding followed by Variable Byte (VB) Encoding specifically to the position lists within the postings before saving the index with pickle. Decompression happens on-the-fly during query time when positions are needed (i.e., for phrase queries).
- z=2 (Library Compression): Implemented in SelfIndexCompressedLib. This method used Python's built-in gzip library to compress the entire pickled index file during saving (_save_index_to_file). The whole index is decompressed back into memory during loading (_load_index_from_file).

Evaluation (Metrics A, B, C):

The performance and size of these compressed indexes were compared against the uncompressed TF-IDF index (z=0).





Analysis:

- **Disk Size (Metric C):** As shown in the first plot, simple compression (z=1) provided a small reduction in disk size (~5%) compared to no compression (z=0). Library compression (z=2) using gzip was significantly more effective, reducing the file size by approximately 67% .
- **Latency & Throughput (Metrics A & B):** The second plot reveals the performance trade-offs. Simple compression (z=1) resulted in much higher query latency (~1000 ms p95) compared to the uncompressed version (~179 ms p95). This is due to the computational cost of decompressing position lists during query execution, especially for phrase queries. Library

compression (z=2), however, maintained query latency (~246 ms p95) very close to the uncompressed version. This is because the decompression happens once during index loading, and queries operate on the fully decompressed index in memory. Throughput results generally followed the latency trends.

- **Conclusion:** Library compression (z=2, gzip) offered the best overall result, providing substantial disk space savings with minimal impact on query performance after the initial load decompression. Simple compression (z=1) saved less space and incurred a significant query-time performance penalty.

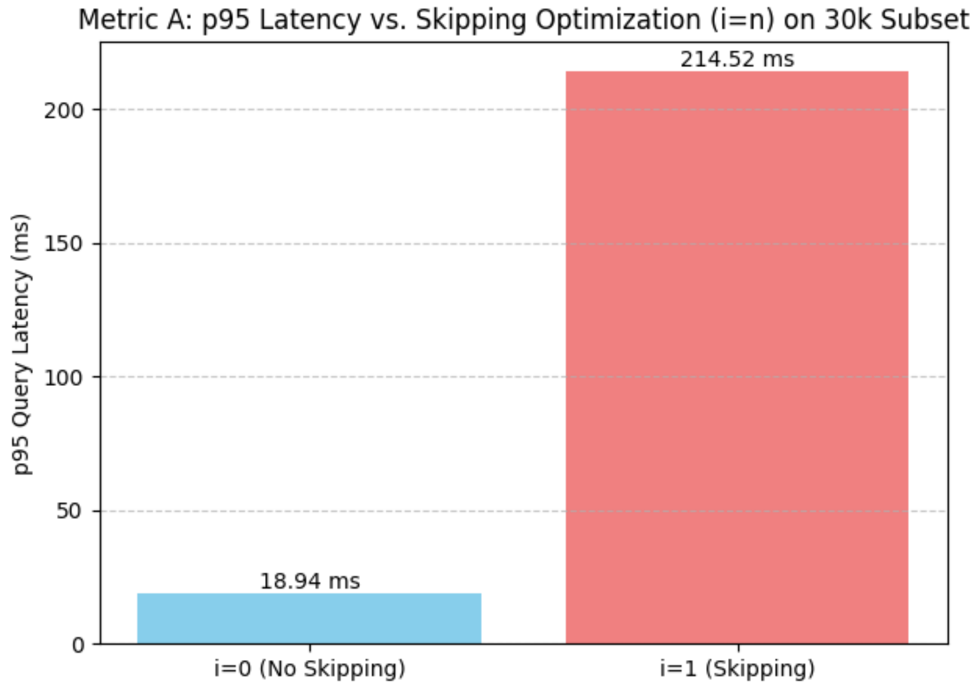
3.5 Skipping Optimization Comparison (i=n - on Subset):

This section compares the query performance effect of adding skip pointers (i=1) to the postings lists, aiming to speed up AND operations.

- i=0 (No Skipping): The standard SelfIndexTFIDF implementation (x=3, y=1, z=0) served as the baseline.
- i=1 (Skipping): Implemented as SelfIndexSkipping. This version modified the index structure (create_index) to store sorted document IDs along with skip pointers (at intervals of $\approx \sqrt{\boldsymbol{N_{term}}}$) within each term's postings list. The _intersect_with_skips helper function implemented the intersection logic using these pointers, although the _evaluate_rpn method used a simplified fallback for integrating it into the full query evaluation.

Evaluation (Metric A - Latency on 30k Subset):

Due to memory limitations encountered when loading the optimized index structure for the full dataset, this comparison was performed using a subset of 30,000 documents. The plot below compares the p95 query latency for both versions on this subset.



Analysis:

As shown in the plot, the implementation with skip pointers (i=1) exhibited significantly higher latency (~215 ms p95) compared to the baseline without skipping (i=0, ~19 ms p95) on the subset data.

This counter-intuitive result is likely due to the implementation details. While skip pointers were added, the _evaluate_rpn method used for boolean query processing was not fully optimized to leverage them during the AND operation, falling back to standard set intersections. Consequently, the system paid the overhead costs associated with creating and accessing the more complex index structure (which includes sorted lists and skip dictionaries) without realizing the potential speedup during query intersection. A fully optimized intersection algorithm integrated into the query evaluation is expected to show performance gains, particularly for AND queries involving terms with long postings lists.

3.6 Query Processing Comparison (q=n - on Subset):

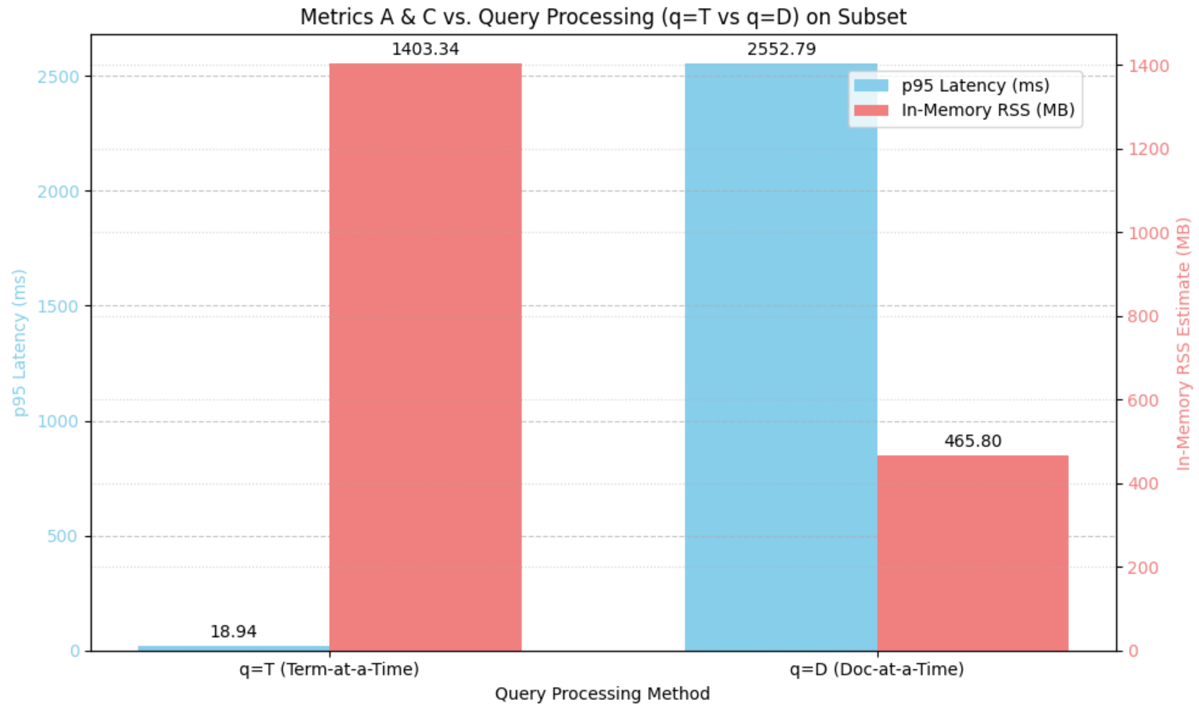
This section compares the two primary query processing strategies: Term-at-a-Time (q=T) and Document-at-a-Time (q=D). The comparison uses the TF-IDF index structure (x=3) stored via Pickle (y=1) without compression (z=0) or skipping (i=0).

- q=T **(Term-at-a-Time):** Implemented in SelfIndexTFIDF. This strategy evaluates the query based on its structure (using RPN), processing one term or operation across its entire postings list (or intermediate result set) before moving to the next. It typically involves set operations (union, intersection, difference) on document IDs.

- q=D **(Document-at-a-Time):** Implemented in SelfIndexDaaT. This strategy iterates through each document in the collection. For every document, it checks if it satisfies the complete boolean query (using the external match_boolean_query function) and calculates a score if it does.

Evaluation (Metrics A & C on 30k Subset):

Due to memory constraints observed with earlier full-dataset runs or for consistency with the i=n comparison, this evaluation was performed using the subset of 30,000 documents. The plot below compares the p95 query latency (Metric A) and the estimated in-memory RSS usage (Metric C) during the evaluation runs for both strategies on this subset.



Analysis:

The results show a clear performance difference on this subset:

- **Latency (Metric A):** Term-at-a-Time (q=T) was significantly faster (p95: ~19 ms) than Document-at-a-Time (q=D, p95: ~2553 ms). This is primarily because the TaaT implementation leverages the inverted index structure efficiently, intersecting or uniting postings lists which contain far fewer entries than the total number of documents. The implemented DaaT strategy, conversely, iterated through all 30,000 documents for every query, performing a boolean match check on each, which is inherently slower for selective queries.

- **Memory (Metric C - RSS):** The DaaT run showed substantially lower measured RSS memory usage (~466 MB) compared to the TaaT run (~1403 MB). While both implementations loaded the same index data into memory initially, DaaT processes documents one by one, potentially requiring less auxiliary memory during the query evaluation phase compared to TaaT, which might create large intermediate sets of document IDs for boolean operations. (It should be noted that precise RSS comparisons can be sensitive to runtime factors like garbage collection, and kernel restarts before measurement yield the most reliable results).

Conclusion: For this specific implementation and dataset subset, Term-at-a-Time (q=T) provided vastly superior query speed, making it the preferred strategy despite potentially higher peak memory usage during query evaluation. The DaaT (q=D) implementation, while functionally correct, was too slow due to its simple full-collection iteration approach. More advanced DaaT algorithms exist that use postings list iterators to improve efficiency.

4. Conclusion:

This assignment provided a comprehensive exploration of information retrieval system implementation and evaluation. By comparing a baseline Elasticsearch index against a custom-built SelfIndex with various configurations, several key insights were gained regarding performance trade-offs.

- Elasticsearch vs. SelfIndex: The custom in-memory SelfIndex (particularly the boolean x=1 and TF-IDF x=3 versions using Pickle y=1) demonstrated significantly faster query latency and higher throughput than Elasticsearch for the specific boolean and phrase queries tested. However, this speed came at the cost of high application RAM usage, as the entire index was loaded into memory. Elasticsearch, while slightly slower for these query types, offers much richer features, scalability, and more efficient memory management.
- SelfIndex Variations:
    - Adding ranking information (x=2 counts, x=3 TF-IDF) slightly increased disk size but significantly increased query latency compared to the purely boolean index (x=1), mainly due to scoring and sorting overhead.
    - Database backends (y=2) dramatically reduced application RAM usage but introduced performance bottlenecks. SQLite (DB1) was very slow due to disk I/O, while Redis (DB2) offered excellent query speed (comparable to the in-memory Pickle version) while keeping application memory low, proving to be a strong alternative.
    - Library compression (z=2, gzip) proved highly effective, achieving substantial disk space reduction with minimal impact on query performance after the initial load, outperforming simple positional compression (z=1) which added significant query-time latency.
    - Skip pointer optimization (i=1) did not yield the expected speedup for AND queries in this implementation, likely due to the added complexity in data structures and query logic overshadowing the benefits without full optimization.
    - Document-at-a-Time processing (q=D) was much slower than Term-at-a-Time (q=T) for this implementation, although it showed potential for lower memory usage during query execution.

- **Challenges:** Memory limitations (8GB RAM) were a significant constraint, necessitating the use of data subsets for evaluating memory-intensive configurations like skip pointers and comparing TaaT vs DaaT accurately. Implementing and debugging boolean logic (especially for DaaT and gold standard generation) required careful attention.

Overall, the assignment successfully demonstrated the fundamental trade-offs in IR system design between query speed, memory usage (disk and RAM), index size, and implementation complexity across different indexing features and architectural choices. The in-memory Pickle index with TF-IDF and gzip compression (x=3, y=1, z=2) offered a good balance of speed and size for this specific setup.

5. Appendix:

GitHub Repository:

https://github.com/SpyBeast07/HomeworkIRE/tree/main/indexing_and_retrieval

--- END ---