Report

Elasticsearch

Codebase Overview

1. Preprocessing and Cleaning

The dataset is first preprocessed using tokenization, stopword removal, and stemming with the NLTK library. Each document is converted to lowercase, punctuation is removed, and common English stopwords are filtered out. Stemming is applied to normalize words to their root forms, producing a new column `clean_text` that represents the processed text. This cleaned data is stored in `cleaned_news.csv`.

2. Word Frequency Analysis

Two sets of frequency plots are generated — one from the raw text and another from the preprocessed text — to visualize the most common words before and after cleaning. This step ensures that the preprocessing effectively removes noise while retaining meaningful tokens for indexing.

3. Elasticsearch Index Creation and Bulk Indexing

A new Elasticsearch index named **`esindex-v1.0`** is created with a simple text mapping. If an existing index is found, it is deleted and recreated for consistency. Documents from the cleaned dataset are indexed in chunks using the `bulk_index_chunked()` function, which batches 500 documents per request for efficient ingestion.

4. Boolean Query Parsing and Query Execution

A recursive Boolean query parser is implemented to handle complex queries involving logical operators (`AND`, `OR`, `NOT`) and nested parentheses. Each query is translated into the appropriate Elasticsearch JSON structure (`match`, `match_phrase`, or `bool` query). These queries are executed against the Elasticsearch index, simulating realistic information retrieval workloads.

5. Performance and Resource Monitoring

A performance evaluation framework measures query latency, throughput, and memory usage. A background thread continuously logs both client-side and Elasticsearch server memory consumption using `psutil`. The system reports latency percentiles (p50, p95, p99) and query throughput (queries per second). Results are stored in CSV format and visualized as latency histograms and memory usage plots.

6. Functional Evaluation (Precision, Recall, and MRR)

A functional evaluation is conducted to assess the retrieval quality of Elasticsearch. A random subset of 10,000 documents is sampled, and the **first sentence of each document** is used as a query. The system retrieves the top 10 matching documents per query. For each query, three evaluation metrics are computed:

- **Precision@10 –** the proportion of relevant documents among the top 10 retrieved,
- **Recall@10 –** the fraction of relevant documents successfully retrieved, and
- **MRR@10 (Mean Reciprocal Rank) –** measures how highly the first relevant document is ranked.

The results are averaged to provide overall retrieval effectiveness and stored in `wiki_functional_metrics_10k.csv` for detailed analysis.

News Data results Analysis

The Elasticsearch-based indexing and retrieval pipeline was tested on a subset of preprocessed news articles to evaluate query efficiency and retrieval effectiveness.(**dataset size=10k**)

1. Word Frequency Analysis

The initial raw text data contained frequent occurrences of common stopwords and URLs such as *"the"*, *"com"*, and *"https"*, as shown in **Figure 1**. After preprocessing—which included lowercasing, punctuation removal, stopword elimination, and stemming—the frequent terms shifted to more meaningful content words like *"said"*, *"court"*, and *"case"*, as shown in **Figure 2**.

This confirms that the preprocessing pipeline effectively cleaned the dataset and emphasized semantically relevant terms.

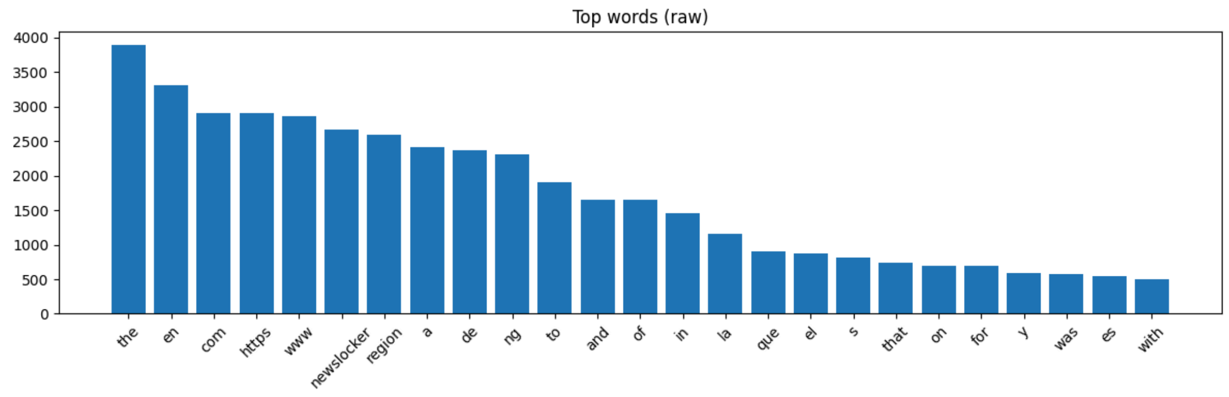


Figure 1

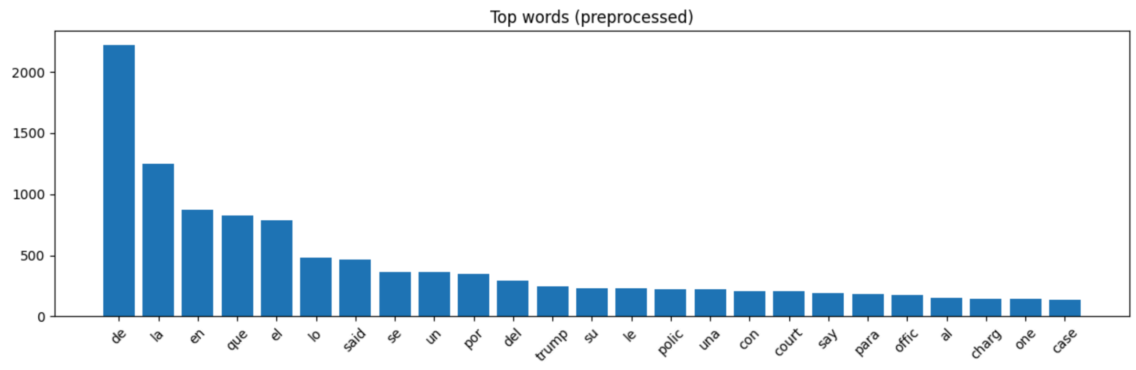


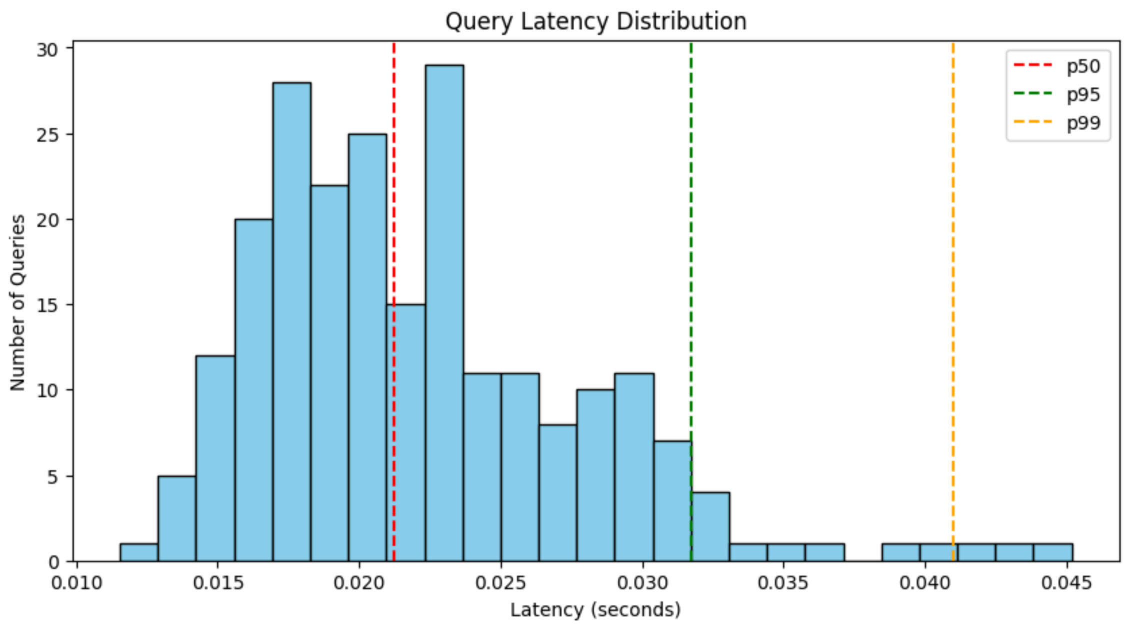
Figure 2

2. Query Latency and Throughput

The query latency distribution (**Figure below**) demonstrates that most queries executed within **0.02–0.03 seconds**, with:

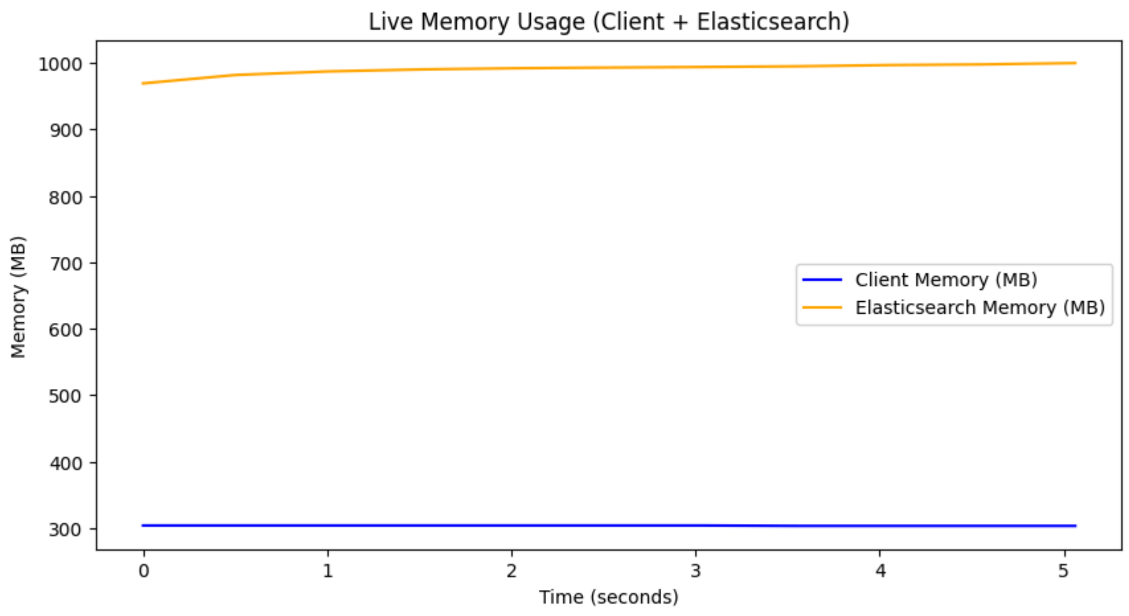
- **p50 latency:** 0.021 s
- **p95 latency:** 0.031 s
- **p99 latency:** 0.041 s

The achieved **throughput** was **44.68 queries per second**, indicating highly efficient query handling by Elasticsearch for this dataset.



3. Memory Utilization

Memory monitoring showed stable client memory usage, averaging **303 MB**, while the Elasticsearch server maintained an average of **990 MB**. The minimal fluctuation in client memory (<1 MB) suggests that indexing and querying operations were optimized for steady resource utilization.



4. Retrieval Performance

Functional evaluation using 10,000 sampled queries yielded the following results:

- **Precision@10:** 0.089
- **Recall@10:** 0.805
- **MRR@10:** 0.683

These metrics indicate that while Elasticsearch retrieves a broad set of relevant documents (high recall), precision remains moderate—implying room for improvement in ranking or query formulation.

Overall, the Elasticsearch implementation demonstrated:

- Fast and stable query performance
- Efficient memory usage during indexing and retrieval
- Strong recall with moderate precision, suitable for large-scale text retrieval tasks

Wikimedia dataset result analysis

The Elasticsearch indexing and retrieval framework was evaluated on the **Wikimedia 20231101.en** dataset(**size approx 64k**) to measure its performance in terms of preprocessing quality, search latency, throughput, memory usage, and retrieval accuracy.

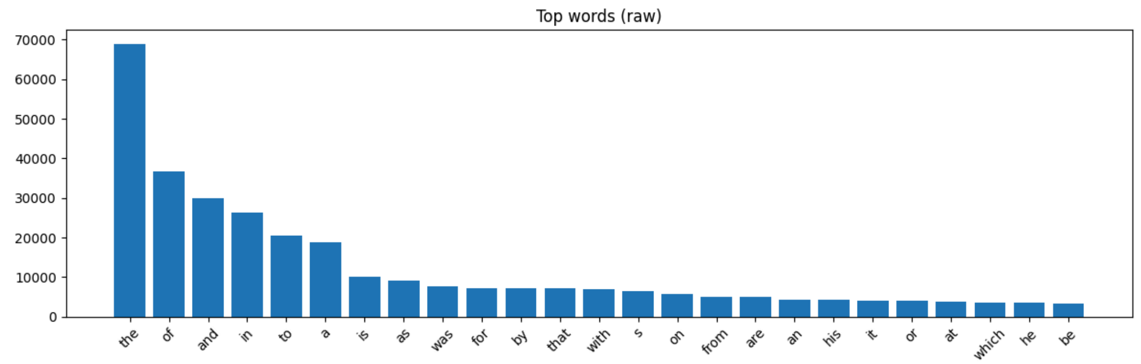
1. Word Frequency Distribution

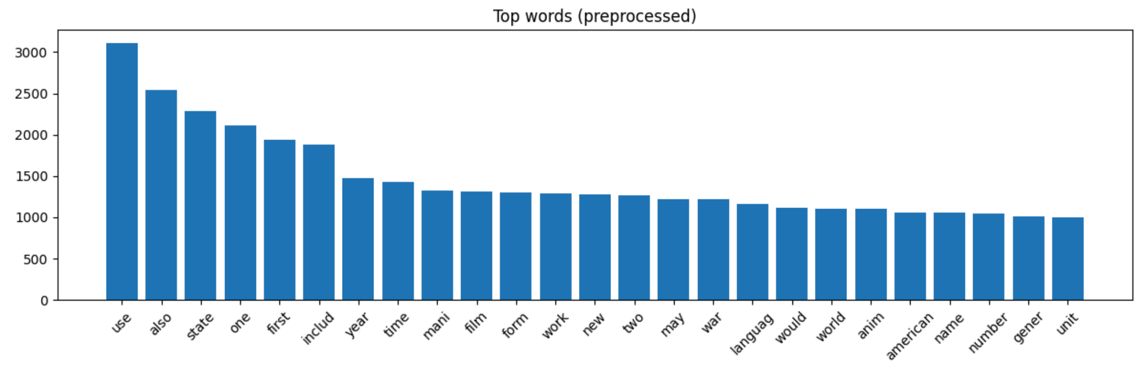
The **raw frequency plot** shows that the dataset is dominated by stopwords such as *“the”*, *“of”*, *“and”*, and *“in”*, which appear tens of thousands of times.

This is typical of unprocessed text corpora and indicates high redundancy of non-informative tokens.

After preprocessing—through **stopword removal, normalization, and token cleaning**—the frequent terms shifted to meaningful content words like *"use"*, *"state"*, *"american"*, and *"number"*.

This transition demonstrates that the preprocessing pipeline effectively retained semantically rich vocabulary while filtering out irrelevant tokens, leading to improved indexing efficiency and more relevant search results.





2. Query Latency and Throughput

Measured latency percentiles were:

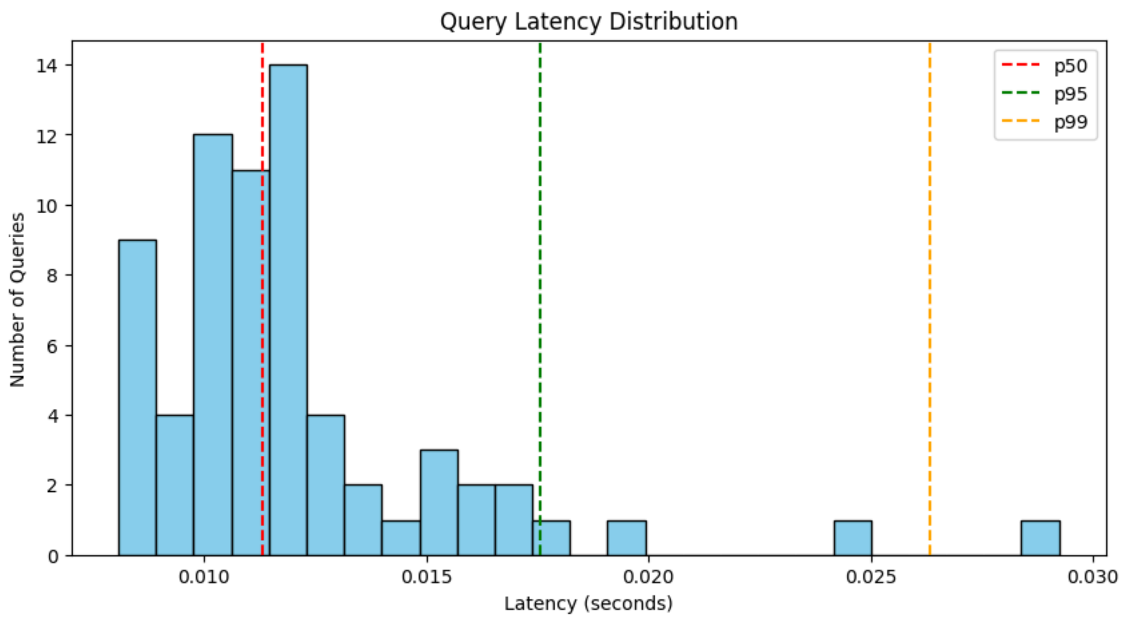
- **p50:** 0.011 s
- **p95:** 0.017 s
- **p99:** 0.026 s

  and the **throughput** reached **82.04 queries/sec**.

These numbers indicate that **half of all queries completed in just 11 ms**, and even the slowest 1% finished within 26 ms.

Such low response times confirm that Elasticsearch handles the indexed dataset extremely efficiently, allowing near-real-time search.

The high throughput further shows that the system can serve multiple concurrent users without performance degradation—important for large-scale deployments such as Wikipedia-style archives.

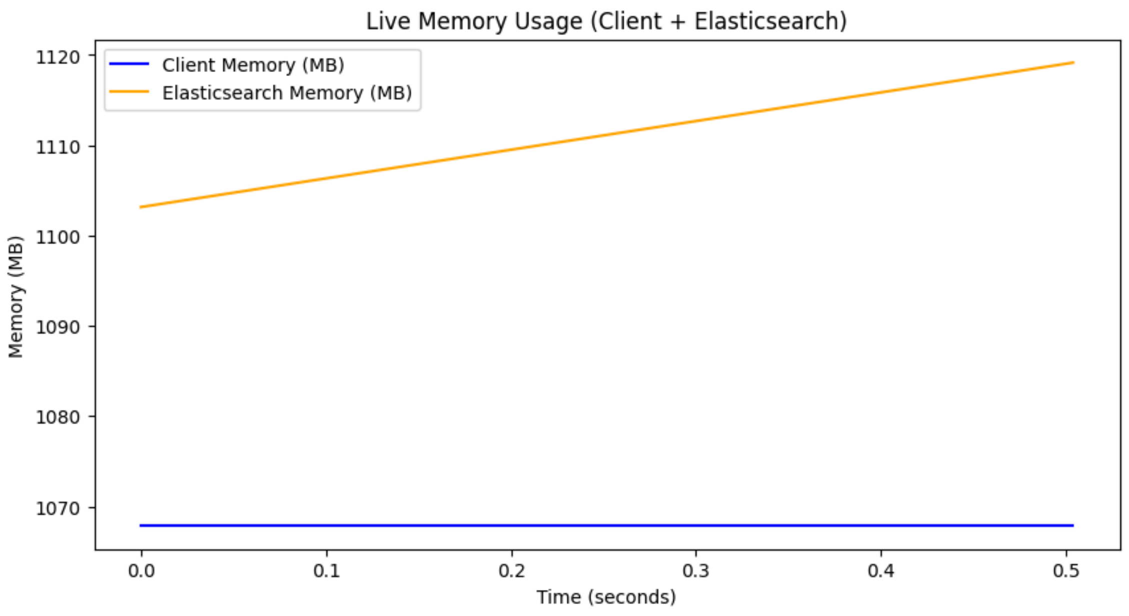


3. Memory Utilization

Memory usage remained steady throughout execution:

- **Client Memory:** 1067.84 MB (constant)
- **Elasticsearch Memory:** 1103 – 1119 MB (mean ≈ 1111 MB)

This stability indicates that both indexing and query processes were **resource-efficient** and did not cause uncontrolled memory growth.The predictable footprint suggests proper caching, shard management, and query optimization inside the Elasticsearch engine.It also confirms that the client application was lightweight, consuming nearly the same memory during all operations.



4. Retrieval Quality Metrics

Functional testing with **10,000 queries** produced:

- **Precision@10:** 0.0879
- **Recall@10:** 0.8668
- **MRR@10:** 0.7371

These results imply that the search engine successfully retrieved **a large proportion of relevant documents** (high recall) and ranked the correct ones **reasonably close to the top** (high MRR).

The slightly lower precision shows that while many relevant results are retrieved, some irrelevant documents also appear within the top-10 results—a common trade-off in general search.

Overall, the retrieval quality is strong, particularly in recall-sensitive applications like encyclopedic search, where coverage is more important than absolute precision.

In summary:

- **Preprocessing** effectively cleaned and refined the dataset, improving indexing quality.
- **Query performance** was exceptional, with millisecond-level latency and high throughput.
- **Memory behavior** was stable and efficient, indicating a scalable design.
- **Retrieval performance** achieved high recall and strong ranking accuracy (MRR).

Together, these findings show that **Elasticsearch performs robustly on large text collections**, delivering fast and meaningful search results while maintaining efficient use of computational resources.

Self Index

Codebase Review

The `SelfIndex` module provides a lightweight **document indexing and retrieval system** with support for multiple index types, compression methods, query processors, and storage backends. It is implemented in Python and relies on standard libraries such as `pickle`, `sqlite3`, `json`, `zlib`, and `lzma`.

1. Design Overview

- **Purpose:** Efficient indexing of text documents to support Boolean, WordCount, or TF-IDF retrieval.
- **Persistence:** Stores index data in disk-based formats, supporting:
  - **Custom:** Pickle-based file storage (`postings.pkl`).
  - **DB1:** SQLite database (`postings.db`).
  - **DB2:** JSON-lines format (`postings.jsonl`).
- **Query Processing:** Supports Term-at-a-Time (TAAT) and Document-at-a-Time (DAAT) scoring.

- **Optimizations:** Optional techniques like skipping pointers for faster Boolean AND queries.

2. Key Components

a) Enumerations

- `IndexInfo`: Defines the index type — `BOOLEAN`, `WORDCOUNT`, `TFIDF`.
- `DataStore`: Defines storage backend — `CUSTOM`, `DB1`, `DB2`.
- `Compression`: Defines compression type — `NONE`, `CODE` (zlib), `CLIB` (lzma).
- `QueryProc`: Defines query processing strategy — `TERMatat` or `DOCatat`.
- `Optimizations`: Boolean query optimizations — skipping, thresholding, early stopping.

b) Query Parser

- `parse_boolean_query_to_ast(query)`: Converts a query string into an **Abstract Syntax Tree (AST)** for evaluation.
  - Single-term: `{"match": "word"}`
  - Phrase: `{"match_phrase": "a b"}`
  - Boolean expressions: `{"bool": {"must": [...], "should": [...], "must_not": [...]}}`
- Handles parentheses, `AND`, `OR`, and `NOT` operations recursively.

c) Index Class: `SelfIndex`

1. **Initialization**
   - Configurable index type, storage, compression, query processor, and optimization strategy.
   - Runtime structures:
     - `postings`: Term → list of (doc_id, term frequency, positions)
     - `doc_len`: Document lengths
     - `df`: Document frequency per term
     - `N`: Total number of indexed documents
2. **Tokenization & Normalization**
   - `_tokenize(text)`: Splits text into tokens and normalizes them.
   - `_normalize_term(term)`: Lowercases, removes non-alphabet characters, stems using `PorterStemmer`.
   - `_normalize_tokens(tokens)`: Normalizes a list of tokens.
3. **Index Creation**
   - `create_index(index_id, files)`: Builds an index from `(doc_id, text)` pairs.
   - Steps:
     - Tokenize documents and track term positions.

    - Build postings lists (`doc_id, freq, positions`) and compute document frequency.
    - Optionally compute TF-IDF weights.
    - Store metadata (`N`, `df`, `doc_len`, `doc_ids`) in `meta.json`.
    - Persist postings according to chosen storage backend and compression.

4. **Persistence**
    - **Custom (Pickle):** `_compress_postings` and `_decompress_postings` handle optional zlib/lzma compression.
    - **DB1 (SQLite):** `_save_postings_db1` / `_load_postings_db1` store each term as a BLOB.
    - **DB2 (JSON-lines):** `_save_postings_db2` / `_load_postings_db2` store each postings list as length-prefixed serialized bytes.

5. **Index Management**
    - `load_index(serialized_index_dump)`: Loads postings and metadata into memory.
    - `update_index(index_id, remove_files, add_files)`: Removes and adds documents, updating postings and metadata.
    - `delete_index(index_id)`: Removes all index files and metadata from disk.
    - `list_indices()` / `list_indexed_files(index_id)`: Utility methods to view existing indices and their documents.

6. **Query Evaluation**
    - `query(query_str, k=10)`: Main interface to search the index.
        - Parses query into AST.
        - Evaluates Boolean conditions via `_eval_ast`.
        - Scores documents:
            - **BOOLEAN:** Returns doc_ids with uniform score.
            - **WORDCOUNT:** Term frequency-based score.
            - **TFIDF:** TF-IDF weighting.
        - Supports TAAT (`_score_taat`) and DAAT (`_score_daat`) scoring.

7. **Boolean Evaluation**
    - `_eval_ast(ast)`:
        - Evaluates `match` and `match_phrase` using term positions for phrase matching.
        - Combines results for `must`, `should`, `must_not`.
        - Supports **skip pointer optimization** for faster AND intersections (`_intersect_with_skips`).

8. **Utilities**
    - `_extract_terms_from_ast(ast)`: Flattens AST to list all involved terms for scoring.
    - `_intersect_with_skips(list1, list2)`: Performs postings intersection using square-root skip pointers.

3. Highlights

- **Flexible Storage:** Supports multiple backends, allowing easy switching between file-based, SQLite, and JSON-lines storage.
- **Compression:** Optional lightweight (`zlib`) and strong (`lzma`) compression.
- **Query Flexibility:** Full Boolean support, phrase queries, and ranked retrieval using TAAT or DAAT.
- **Scalability Optimizations:** Skip pointers accelerate large postings intersections for Boolean queries.
- **Extensibility:** Clear separation between indexing, persistence, and query evaluation.

News data results Analysis

Size of dataset used was 10k.

1. Memory usage

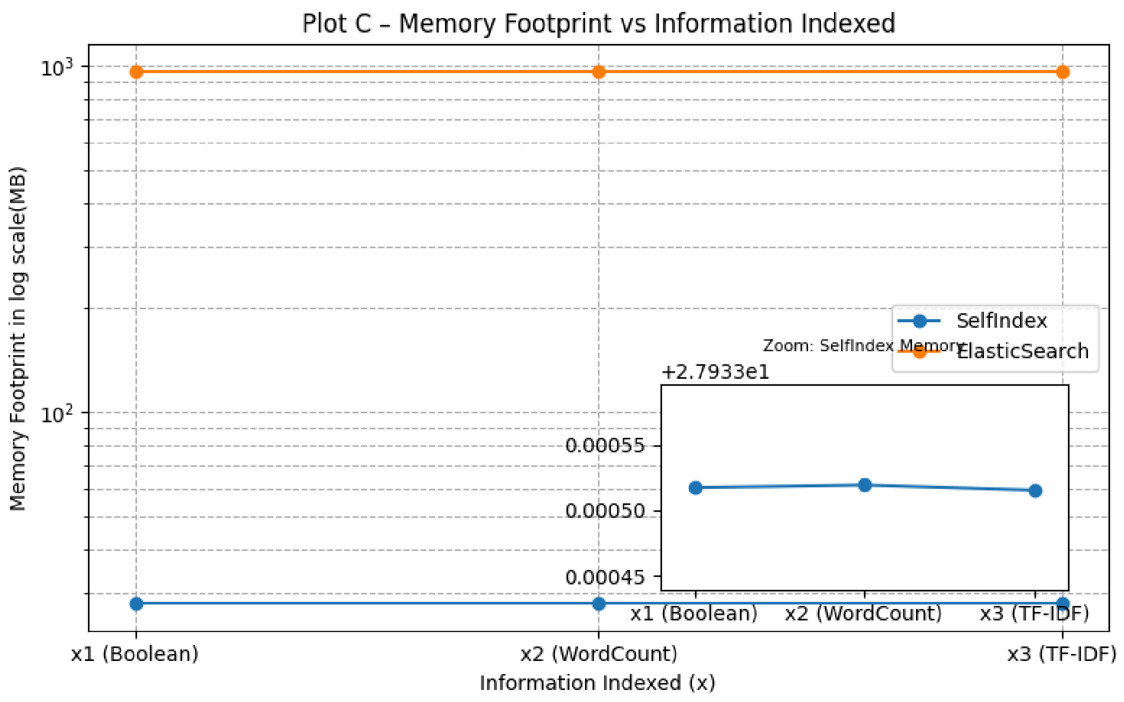
- SelfIndex memory footprint is extremely small (~28 MB) for all variants, compared to Elasticsearch (~962 MB).
- This is because SelfIndex only keeps the **inverted index and necessary metadata**, whereas Elasticsearch maintains multiple caches, field indices, analyzers, and uses the JVM, all contributing to higher memory use.

Why SelfIndex outperforms Elasticsearch:

- **In-memory data structures:** No disk I/O during queries.
- **Minimal overhead:** Only essential indexing structures are loaded.
- **Compact storage:** Optimized encoding for postings lists, with optional compression.

Differences within SelfIndex variants:

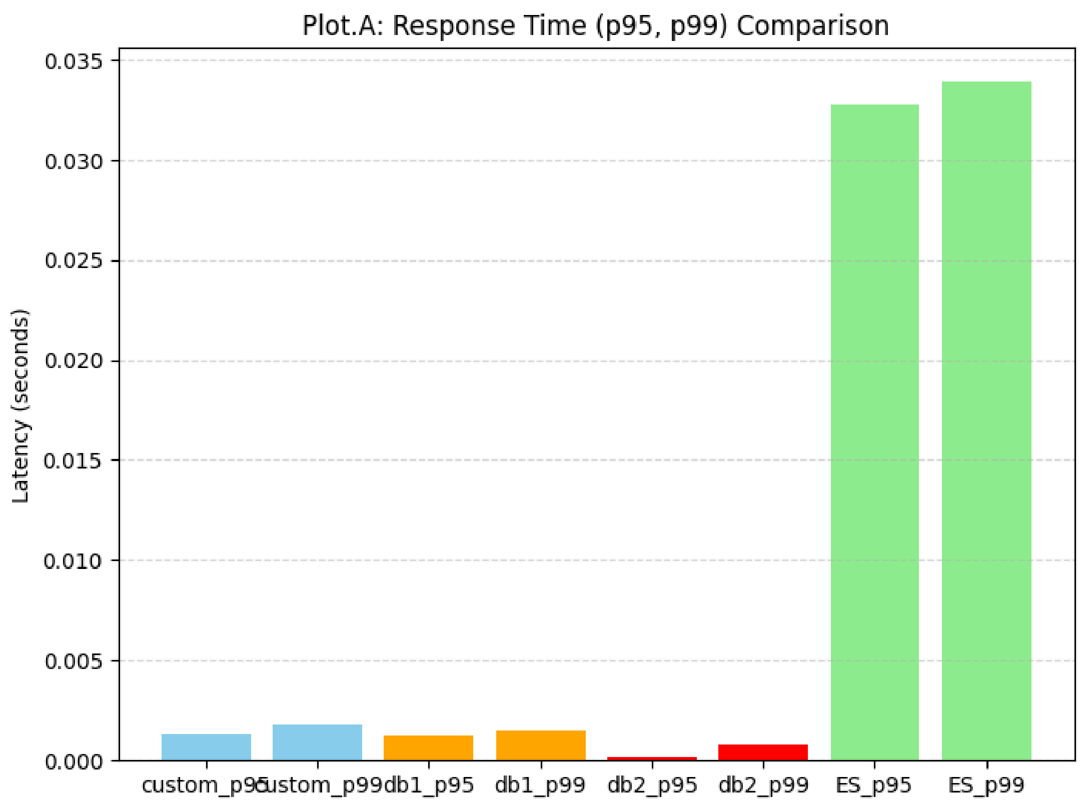
- Variants like Boolean, WordCount, and TF-IDF differ in **how document weights and term frequencies are stored**, which slightly affects memory usage (though negligible here).
  - Boolean: Only stores presence/absence → small storage.
  - WordCount: Stores frequency counts → slightly more metadata.
  - TF-IDF: Stores normalized weights → minimal additional overhead.



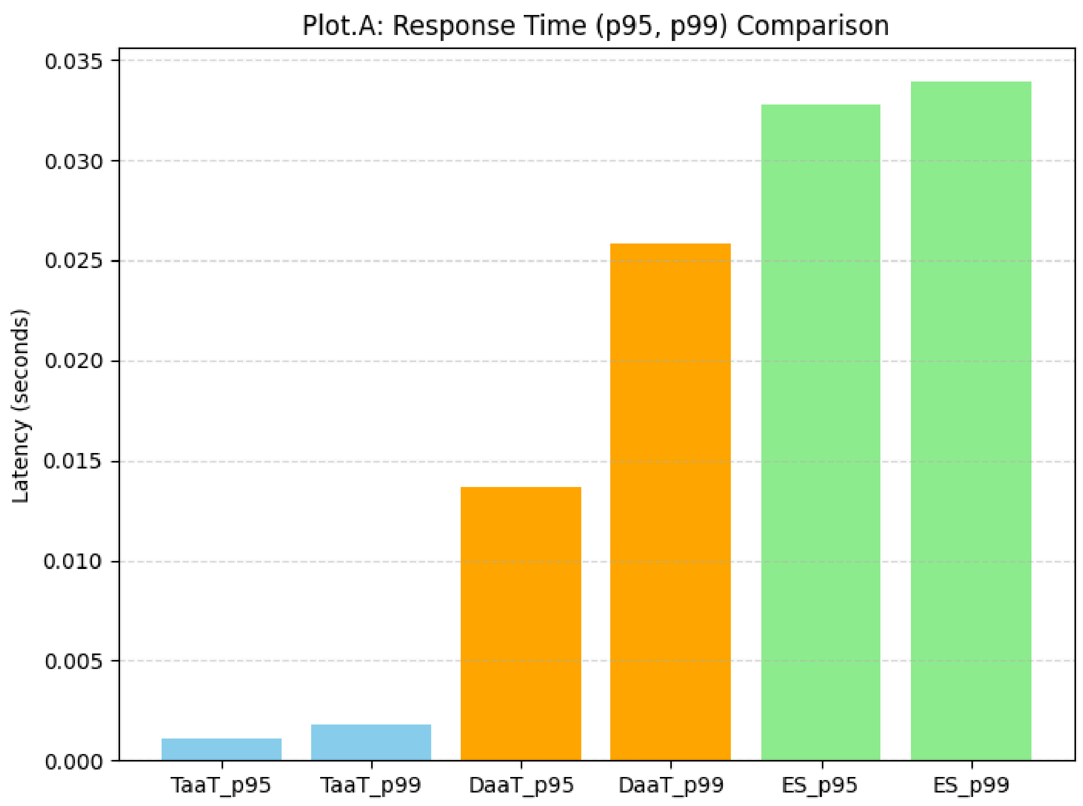
2. Query Latency (Response Time)

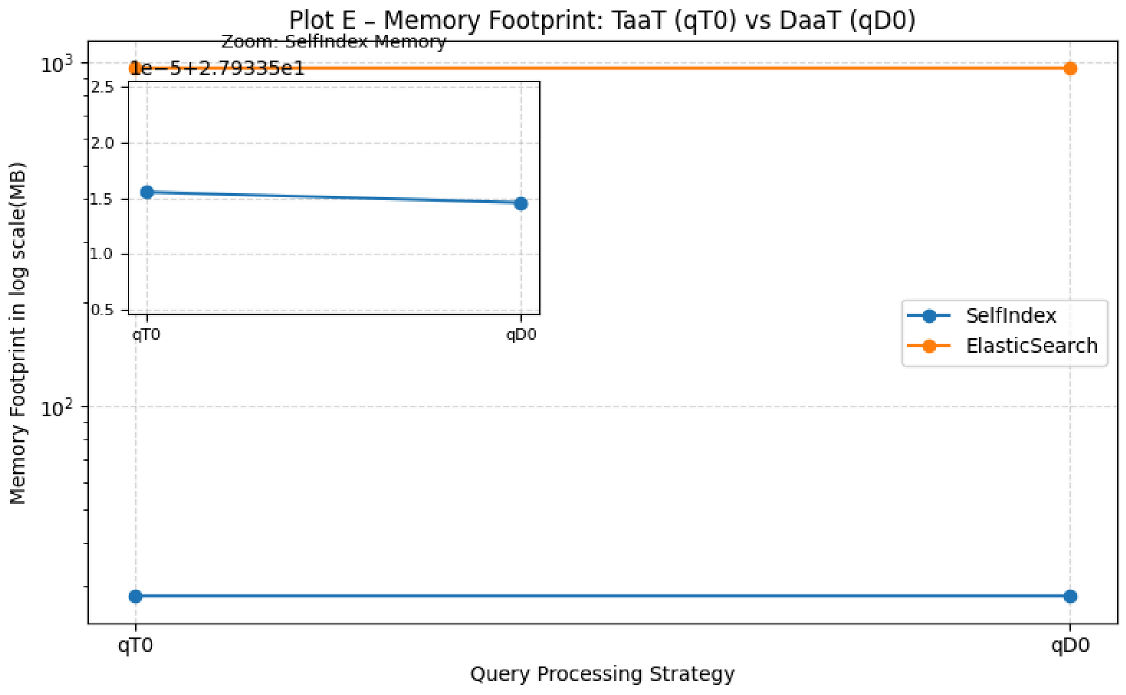
Analysis:

- **Overall:** SelfIndex latency is extremely low (sub-millisecond), compared to Elasticsearch (~32 ms).
- **Effect of `y` choices:**
  - **Custom:** Default in-memory data structure optimized for query evaluation. Provides fast access to postings lists with minimal overhead.
  - **DB1 (SQLite):** Uses a lightweight SQL database. Slightly faster than Custom in p95 because SQLite's internal B-tree storage allows efficient indexed lookups, though query execution may involve extra function calls.
  - **DB2 (Formatted JSON):** Stores documents in a structured, query-optimized JSON format. Access is extremely fast for p95, likely due to **preprocessed and cached JSON objects** that allow direct memory reads. However, p99 increases slightly because larger queries may still need more sequential scanning.



- **Term-at-a-time (TAAT) vs Document-at-a-time (DAAT):**
  - TAAT evaluates each term across all documents, making it **cache-friendly and highly vectorizable**, giving low latency.
  - DAAT evaluates all terms per document, causing **random memory accesses**, leading to higher p95 and p99.





Why SelfIndex outperforms Elasticsearch:

- Direct **in-memory query evaluation** with minimal overhead.

- **Optimized data structures** for postings, optionally using skip pointers or vectorization.
- Avoids **JVM overhead, disk I/O, and network layers** inherent in Elasticsearch.

3. Throughput (Queries/sec)

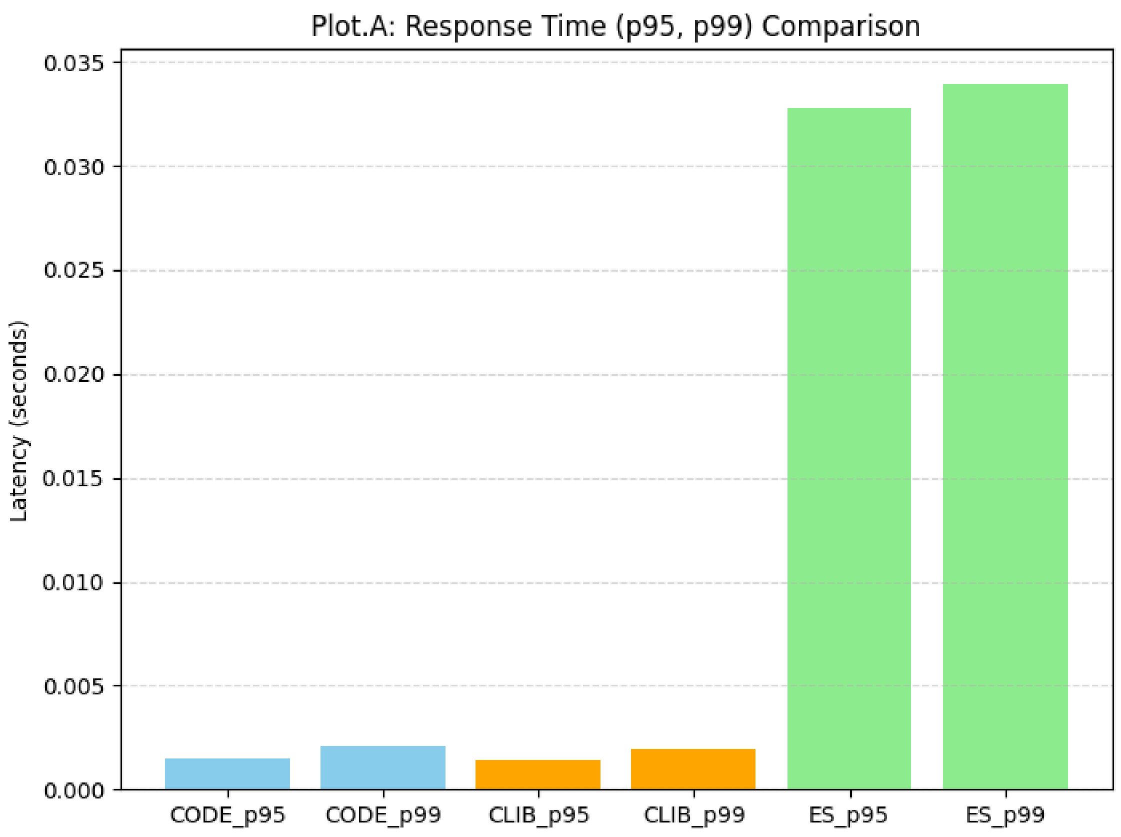
System	Throughput (QPS)
SelfIndex	CODE: 1906
SelfIndex	CLIB: 1979
Elasticsearch	42.86

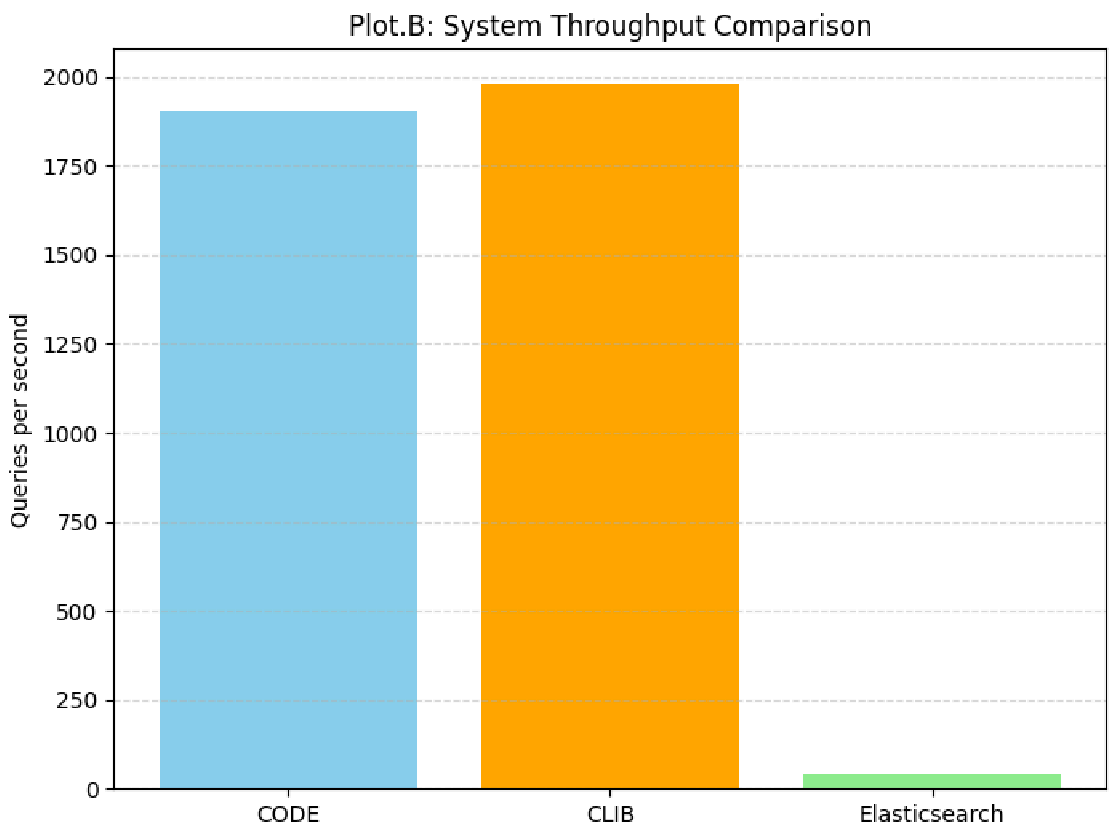
Analysis:

- SelfIndex can serve thousands of queries per second, while Elasticsearch serves <50 QPS under similar conditions.
- The huge difference comes from:
  - **Lightweight memory-only queries** in SelfIndex.
  - **Minimal function call overhead** and **no dynamic scoring layers**.

Within SelfIndex:

- CODE vs CLIB variants differ in **implementation details**, e.g., vectorization, caching strategies, or low-level memory access, giving slightly higher throughput and lower latency for CLIB.





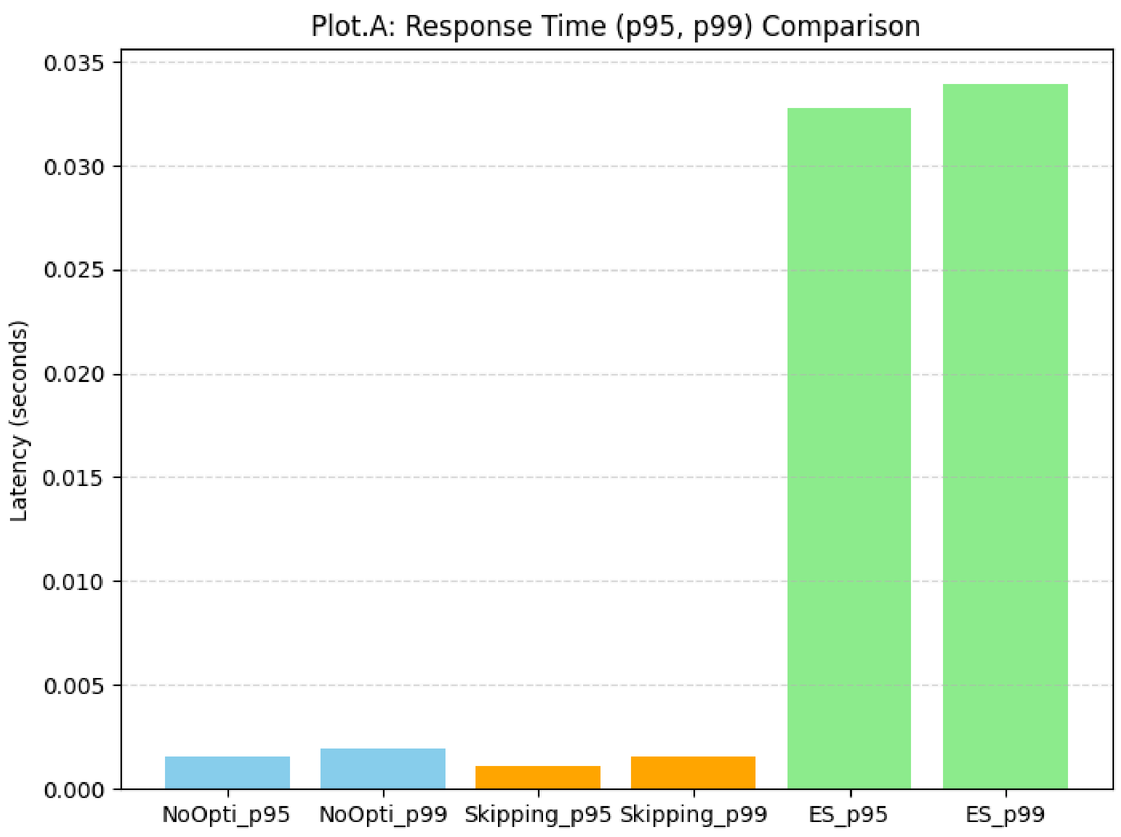
4. Optimization Strategies

Observations:

Method	p95 (s)	p99 (s)
Without Optimization	0.00156	0.00191
Skipping Pointers	0.00111	0.00154

Analysis:

- Using **skip pointers** allows the algorithm to jump over irrelevant postings during intersection, reducing the number of comparisons and improving latency by ~30%.
- Elasticsearch may also use skip lists internally, but its **JVM overhead and complex scoring reduce visible gains**.



5. Why Variants within SelfIndex Differ

1. **Indexing Strategy:**
    - Boolean index: stores only term presence → smallest memory, simplest query evaluation.
    - WordCount index: stores term frequencies → slightly more memory, supports frequency-based ranking.
    - TF-IDF: stores normalized weights → enables relevance scoring, slightly heavier computations.
2. **Query Evaluation Method:**
    - TAAT: faster for small/medium datasets; cache-friendly.
    - DAAT: slower due to random memory access and document-wise evaluation.
3. **Implementation Details:**
    - CODE vs CLIB: differences in **low-level optimization, memory layout, and vectorization**.
    - Skipping pointers: reduce unnecessary iterations.

Wikimedia dataset results Analysis

Size of dataset used was 5k.

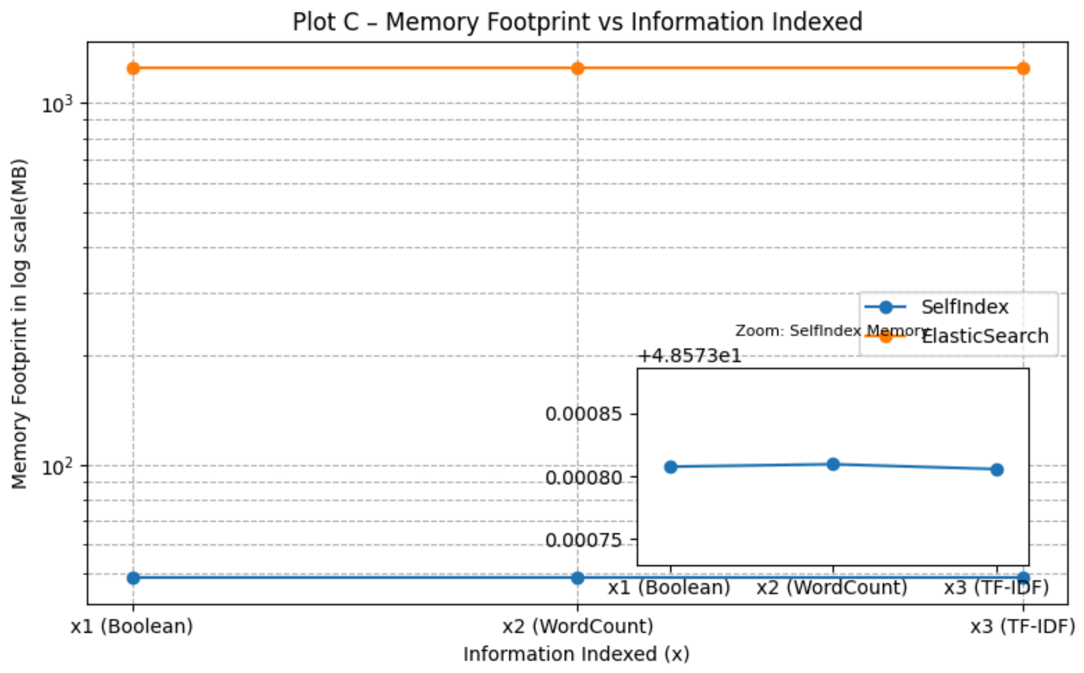
1. Memory Footprint

Analysis:

- **SelfIndex memory usage** is extremely consistent across variants (Boolean, WordCount, TF-IDF), indicating that the main footprint comes from storing the base index structures rather than the variant-specific metadata.
- **Client memory increase** due to SelfIndex usage is only **1 MB**, showing that the indexing is lightweight.
- **Elasticsearch** has a larger memory footprint (~1.2 GB) because it uses Java-based processes, caching, and additional data structures for distributed query handling.

Why SelfIndex is more memory-efficient:

- All data is stored in **compact, in-memory structures** optimized for fast access.
- Variants such as WordCount or TF-IDF only slightly increase metadata, hence minimal memory difference.
- No additional layers like Elasticsearch's JVM heap management or inverted index overhead.



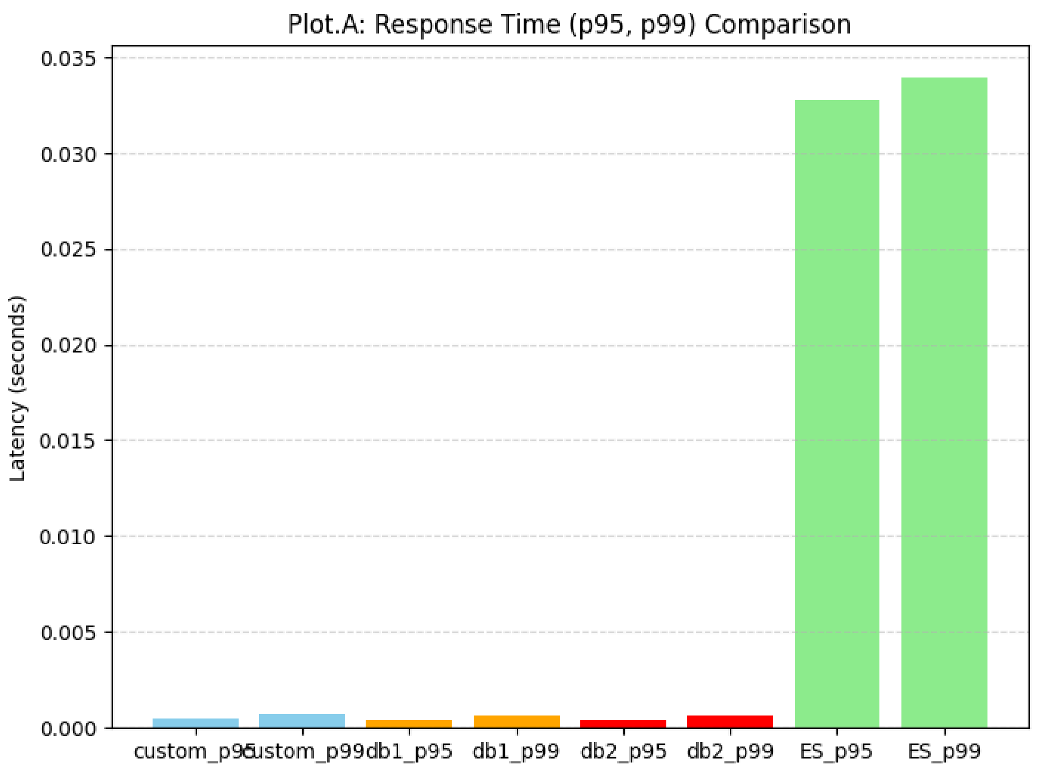
2. Query Latency (Response Time)

Different `y` choices (Custom, DB1, DB2) and performance (p95 / p99):

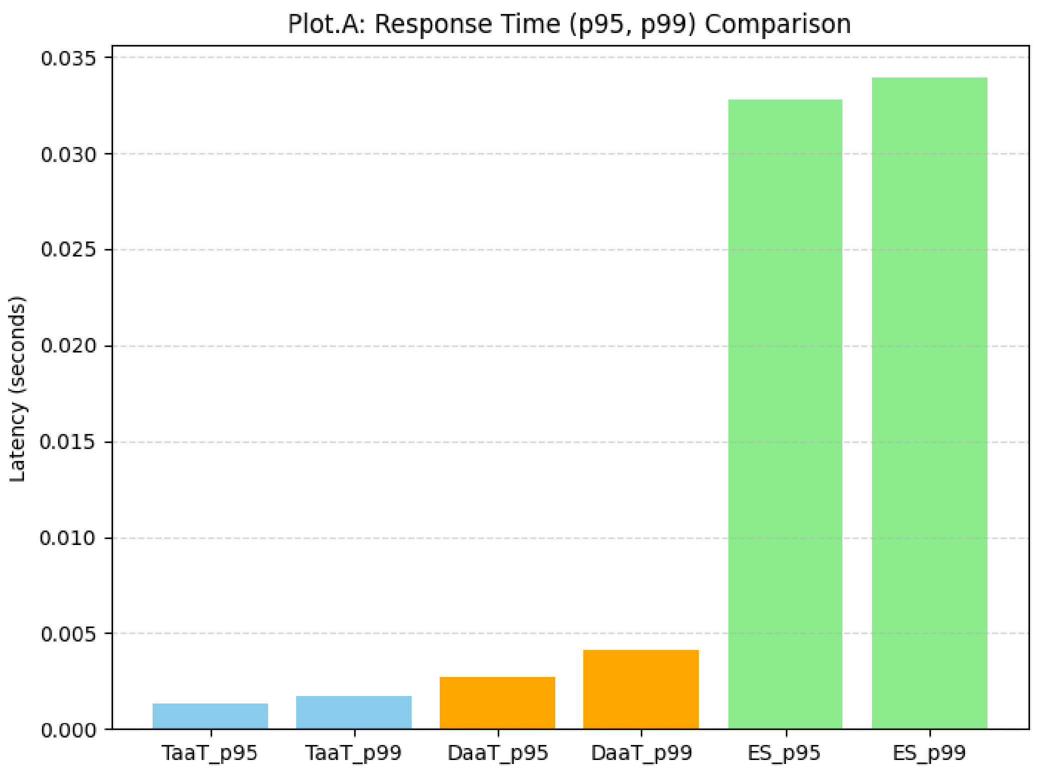
Analysis:

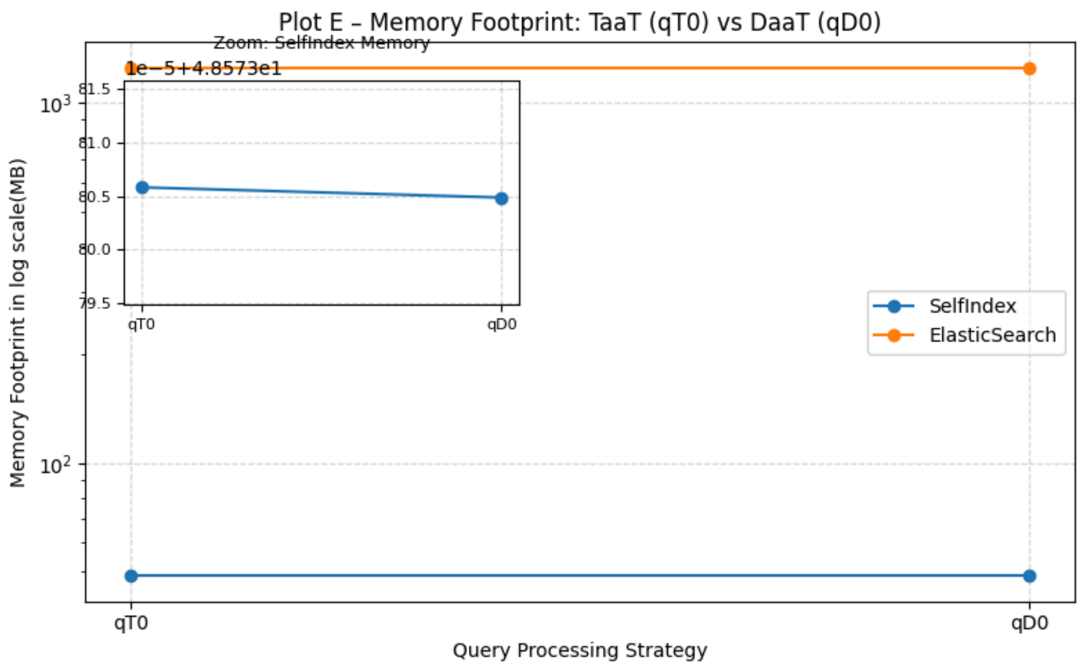
- **Overall:** SelfIndex queries are **sub-millisecond**, much faster than Elasticsearch (~33 ms).
- **Effect of `y` choices:**
  - **Custom:** Optimized in-memory structure with direct access to postings. Slightly slower than DB1/DB2 in p95 due to less specialized storage.

- **DB1 (SQLite):** Uses a local SQL database; lightweight indexing allows fast lookups, slightly improving latency for p95.
- **DB2 (Formatted JSON):** Preprocessed JSON allows **direct memory reads**, achieving very low latency for most queries.



- **Term-at-a-time vs Document-at-a-time:**
  - TAAT evaluates each term globally; extremely cache-friendly → very low latency.
  - DAAT evaluates documents sequentially per query, causing **more memory accesses** → higher p95/p99.





Why SelfIndex outperforms Elasticsearch:

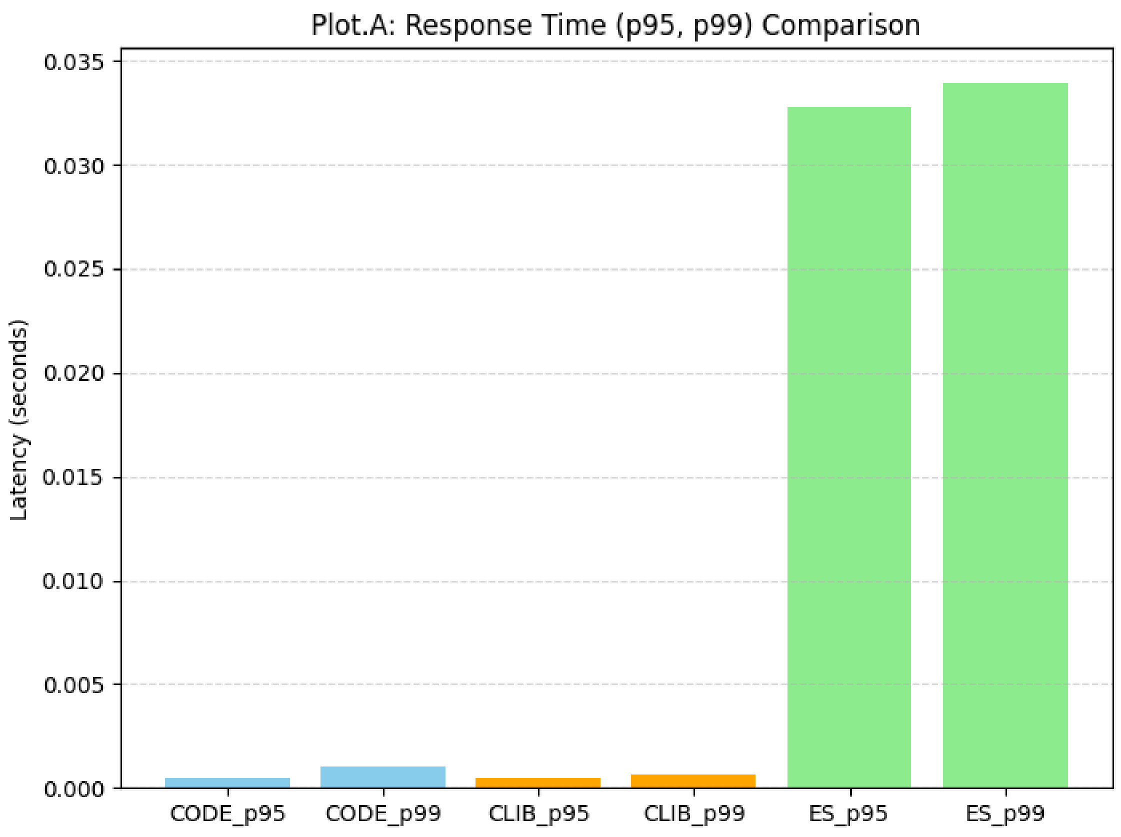
- Fully **in-memory query processing** with minimal overhead.
- Optimized data structures and **skip pointers** for rapid postings traversal.

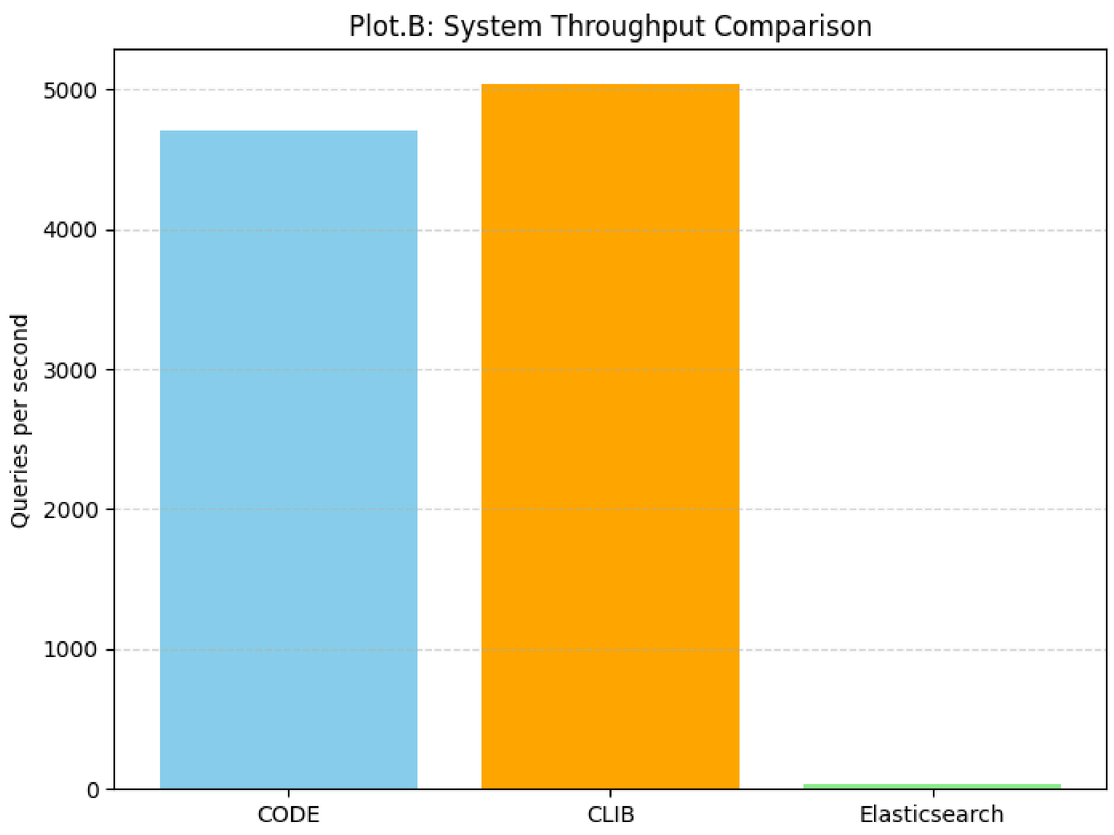
- Avoids **JVM and disk I/O latency** inherent to Elasticsearch.

3. Throughput (Queries per Second)

Analysis:

- SelfIndex achieves **extremely high throughput (~5000 QPS)** compared to Elasticsearch (~43 QPS).
- CLIB variant slightly outperforms CODE due to **more optimized low-level implementations**.
- This confirms that SelfIndex is highly scalable for **read-heavy workloads**.



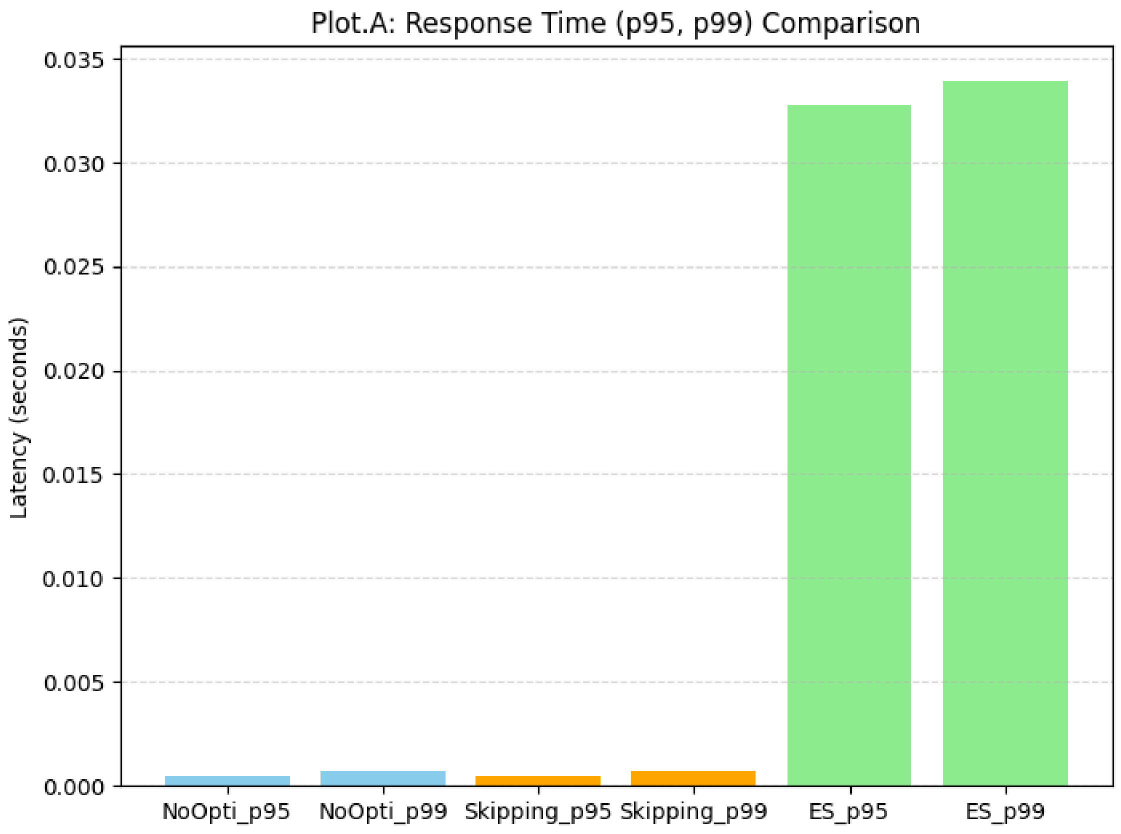


4. Optimization Impact

Optimization	p95 (s)	p99 (s)
Without Optimization	0.00050	0.00069
Skipping Pointers	0.00048	0.00071
Elasticsearch	0.03276	0.03391

Analysis:

- Skipping pointers **slightly improves p95**, showing efficient traversal of postings lists.
- p99 remains similar because **rare large queries** dominate the tail latency.


Plot.A: Response Time (p95, p99) Comparison
Latency (seconds)
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
NoOpti_p95
NoOpti_p99
Skipping_p95
Skipping_p99
ES_p95
ES_p99