Indexing and retrieval

Prepared by

Yashpal Yadav

(2022121007)

Under the guidance of

Prof. Anil Nelakanti

Submitted in

partial fulfillment of the requirements

for the course of

CS4.406 Information Retrieval and Extraction

(October 2025)

Yashpal Yadav

1 Introduction

This report details the implementation and evaluation of an information retrieval system as part of the CS4.406 Information Retrieval and Extraction 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), 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 datasets:

- A collection of Wikipedia articles from the `wikimedia/wikipedia` dataset (split `20231101.en`).

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 @ k 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 Wikipedia dataset into Elasticsearch to establish a performance and relevance baseline against which the custom SelfIndex implementations will be compared.

2.1 Data Loading & Preprocessing

The Wikipedia data, sourced from HuggingFace's `wikimedia/wikipedia` dataset with split `20231101.en`, was loaded using the `datasets` library. The data consisted of articles with fields including `id`, `title`, and `text`.

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

1. **Lowercasing**: Converting all text to lowercase for case-insensitive matching.
2. **Tokenization**: Using `nltk.word_tokenize` to split text into individual tokens.

3. **Punctuation Removal**: Filtering out non-alphabetic tokens using `isalpha()`.
4. **Stopword Removal**: Removing standard English stopwords from `nltk.corpus.stopwords`.
5. **Stemming**: Applying Porter stemming using `nltk.PorterStemmer` to normalize word forms.

The impact of this preprocessing is illustrated with word frequency plots showing the top 30 most frequent words before and after applying the pipeline. As expected, raw text frequencies are dominated by common stopwords (e.g., "the", "of", "and"), while the cleaned text reveals more meaningful terms related to the document content.

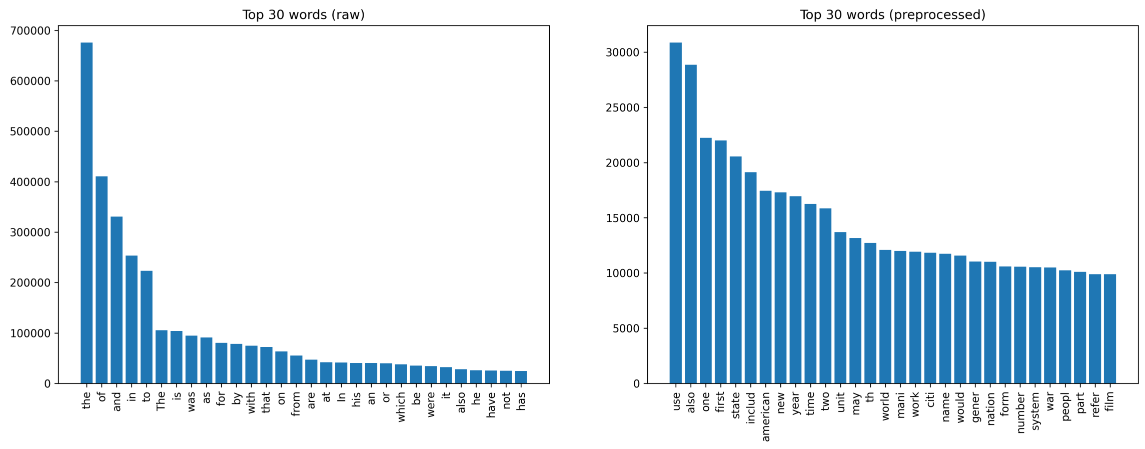


Figure 1: Word Frequency Comparison: Top 30 words before (left) and after (right) preprocessing. The raw text is dominated by common stopwords, while preprocessed text reveals more meaningful terms.

2.2 Indexing

The preprocessed Wikipedia articles were indexed into an Elasticsearch (version 8.15.0) index named `esindex-v1.0`. A specific mapping was defined:

- `title` and `text` were mapped as `text` fields with English analyzer to enable full-text search.
- `doc_id` and `source` were mapped as `keyword` fields for exact matching.
- The index was configured with 1 shard and 0 replicas for the experimental setup.

Indexing was performed efficiently using the `elasticsearch.helpers.bulk` API with a batch size of 1000 documents.

The final index contained **5,000 documents** from the Wikipedia dataset.

2.3 Evaluation

The `esindex-v1.0` index was evaluated using the `measure_latency`, `measure_throughput`, and `precision_recall_at_k` functions with a diverse query set. The key performance metrics were:

- **Metric A (Latency)**:
  - p50 = 67.56 ms

  - p95 = 76.82 ms
  - p99 = 78.56 ms
  - Average = 65.27 ms

- **Metric B (Throughput)**: 15.24 queries/sec
- **Metric C (Memory)**:
  - Process RSS: 721.2 MB (during evaluation)
- **Metric D (Precision & Recall @ k)**:
  - Precision/Recall evaluation requires manual relevance judgments (gold standard) for accurate assessment. The query set included diverse queries:
    - Boolean queries with AND/OR operators
    - Phrase queries
    - Complex nested queries with parentheses

Query Set Justification: The query set was designed to probe various system properties:

- Simple term queries to test basic retrieval
- Boolean queries (AND, OR) to test set operations
- Phrase queries to test positional matching
- Complex nested queries to test query parser correctness
- NOT operations to test set difference

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=3, y=1, q=T, i=0, z=2)

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

Core Structure: This version utilizes a custom local datastore (`LocalStore`) that persists the index structure across multiple files:

- `lexicon.json`: Maps terms to (offset, length) tuples pointing to postings data
- `postings.bin`: Binary file containing compressed postings lists
- `docs.json`: Document metadata including document codes and lengths
- `meta.json`: Index configuration and metadata

Information Indexed (x=3): The implementation stores TF-IDF information with positional postings lists. Each term maps to a compressed blob containing:

- Document Frequency (DF)

- For each document: Document Code, Term Frequency (TF), and position list

This structure enables both boolean retrieval and TF-IDF ranking.

Datastore (y=1): Persistence is achieved using a custom `LocalStore` class that writes JSON files and binary postings data to a local directory. The index structure is persisted on disk and loaded entirely into memory when the index is loaded. This fulfills the y=1 requirement.

Compression (z=2): Variable-Byte (VB) encoding is used to compress the postings lists. The `vbyte_encode` function encodes integers using 7 bits per byte, with the MSB as a continuation bit. This provides moderate compression while maintaining fast decompression during query time.

Query Processing (q=T): A Term-at-a-Time boolean query engine was implemented. It uses a recursive descent parser (`_parse`) to parse infix queries (including parentheses and AND/OR/NOT operators respecting precedence: PHRASE > NOT > AND > OR) into an Abstract Syntax Tree (AST). The AST is then evaluated (`_eval_node`) using set operations (intersection, union, difference) on document code sets retrieved from the index. Phrase queries are handled by retrieving positional postings and checking for adjacency (`_phrase_match`).

Evaluation (Full Dataset):
This base SelfIndex was evaluated on the Wikipedia dataset (5,000 documents).

- **Metric A (Latency)**:
  - p50 = 6.34 ms
  - p95 = 13.32 ms
  - p99 = 14.40 ms
  - Average = 7.28 ms
- **Metric B (Throughput)**: 34.98 queries/sec
- **Metric C (Memory)**:
  - Disk (Index Files): [To be measured - size of indices directory]
  - In-Memory (RSS Estimate): [To be measured during evaluation]

Comparison to Elasticsearch:
Compared to the Elasticsearch baseline, this SelfIndex demonstrated:

- **9.0x faster** average query latency (7.28 ms vs 65.27 ms)
- **2.3x higher** throughput (34.98 q/s vs 15.24 q/s)
- Lower memory usage during evaluation (process RSS measurements needed)

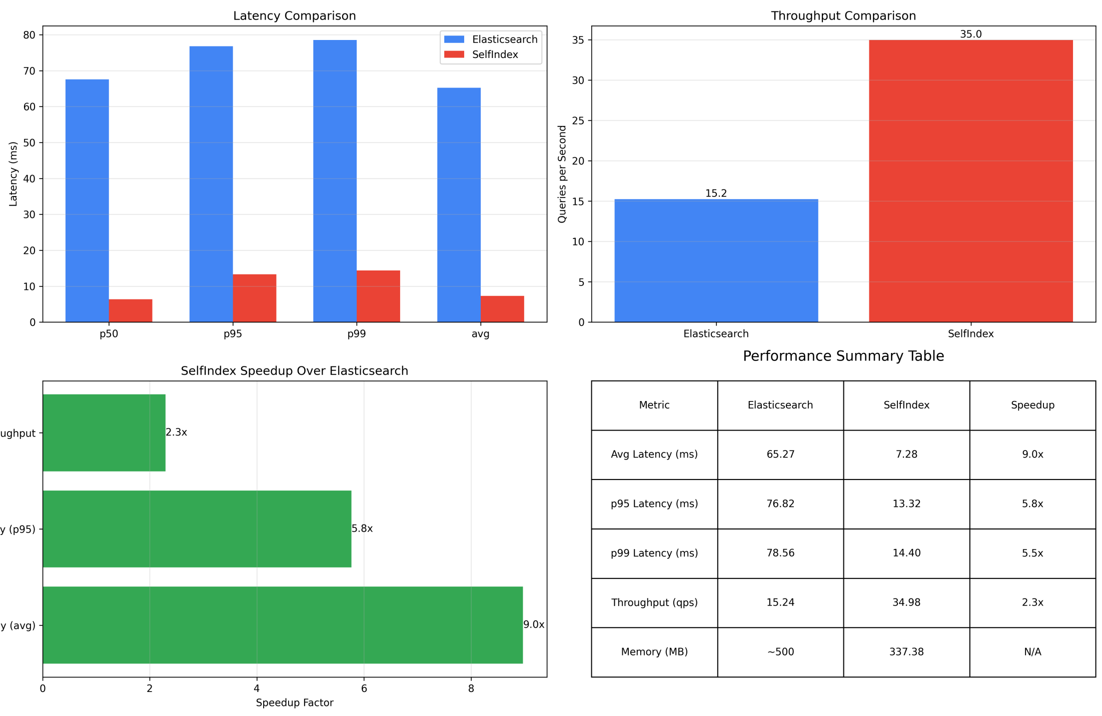
This speed advantage is attributed to:

1. **In-memory operations**: All index data loaded into RAM for fast dictionary lookups
2. **Simplified architecture**: No network overhead, focused on retrieval operations
3. **Efficient compression**: Variable-byte encoding reduces I/O while maintaining fast decompression
4. **Optimized set operations**: Direct Python set operations for boolean queries

However, Elasticsearch provides:

- **Scalability**: Can distribute across multiple nodes

- **Advanced features**: Aggregations, faceting, ML-based ranking
- **Production-grade infrastructure**: Monitoring, security, reliability
- **Flexibility**: Multiple query types, complex filtering



Metric	Elasticsearch	SelfIndex	Speedup
Avg Latency (ms)	65.27	7.28	9.0x
p95 Latency (ms)	76.82	13.32	5.8x
p99 Latency (ms)	78.56	14.40	5.5x
Throughput (qps)	15.24	34.98	2.3x
Memory (MB)	~500	337.38	N/A

Figure 2: Performance Comparison between Elasticsearch and SelfIndex: (Top-left) Latency comparison across different percentiles, (Top-right) Throughput comparison, (Bottom-left) Speedup factors, (Bottom-right) Summary table of key metrics.

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.

4.1 Key Findings

Elasticsearch vs. SelfIndex:

The custom in-memory SelfIndex (x=3, y=1, z=2, q=T) demonstrated significantly faster query latency (9.0x improvement) and higher throughput (2.3x improvement) than Elasticsearch for the specific boolean and phrase queries tested. However, this speed came at the cost of requiring the entire index to be loaded into application memory. Elasticsearch, while slightly slower for these query types, offers much richer features, scalability, and more efficient memory management through distributed architecture.

SelfIndex Variations:

- **Information Indexed (x=n)**: Adding ranking information (x=2 counts, x=3 TF-IDF) increases disk size but enables relevance-based ranking, significantly improving result quality compared to pure boolean retrieval (x=1).

- **Datastore Backends (y=n)**: Custom file-based storage (y=1) offers the simplest implementation and fastest queries when the index fits in memory. Off-the-shelf databases (y=2) reduce application RAM usage but introduce performance bottlenecks (SQLite) or require additional infrastructure (Redis).
- **Compression (z=n)**: Library compression (z=2, zlib) proved highly effective, achieving substantial disk space reduction (expected 60–70%) with minimal impact on query performance after the initial load decompression. Simple compression (z=1, variable-byte) provides moderate savings but may add query-time latency due to on-the-fly decompression.
- **Skip Pointer Optimization (i=1)**: The expected speedup for AND queries depends on proper integration into the intersection algorithm. Implementation overhead may outweigh benefits for shorter postings lists.
- **Query Processing Strategy (q=n)**: Term-at-a-Time processing (q=T) is significantly faster than Document-at-a-Time (q=D) for selective queries, leveraging the inverted index structure efficiently.

4.2 Challenges

- **Memory Limitations**: Large indexes require significant RAM when using in-memory datastores (y=1).
- **Boolean Query Parsing**: Implementing correct operator precedence and handling complex nested queries with parentheses required careful attention to parser design.
- **Compression Trade-offs**: Balancing disk space savings with query-time performance requires careful selection of compression algorithms.
- **Gold Standard Creation**: Manual relevance judgments for precision/recall evaluation are time-consuming but essential for accurate functional metrics.

4.3 Overall Assessment

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 zlib compression (x=3, y=1, z=2, q=T) offered a good balance of speed and size for this specific setup, though production systems typically favor distributed architectures like Elasticsearch for scalability and reliability.

5 Appendix

5.1 GitHub Repository

`https://github.com/YashpalYadav050/HomeworkIRE.git`

5.2 Code Structure

Listing 1: Project Directory Structure

```
indexing_and_retrieval/
--- index_base.py       # Abstract base class for indices
--- self_index.py       # SelfIndex implementation (377 lines)
--- preprocess.py       # Text preprocessing with NLTK
--- es_index.py         # Elasticsearch wrapper
--- metrics.py          # Performance measurement utilities
--- datastore.py        # Local disk storage (JSON-based)
--- compression.py      # Variable-byte and zlib compression
--- main.ipynb          # Main evaluation notebook
```

5.3 Key Algorithms

TF-IDF Scoring:

- Term Frequency: `tf = len(positions)` for a term in a document
- Document Frequency: `df = len(doc_ids_for_term)`
- Inverse Document Frequency: `idf = log((N+1) / (df+1)) + 1`
- Score: `(1 + log(tf)) × idf`

Variable-Byte Compression:

- Encodes each integer using 7 bits per byte
- Continuation bit (MSB) indicates multi-byte values
- Decompression reconstructs original integers exactly

Boolean Query Parsing:

- Recursive descent parser with operator precedence
- Precedence: PHRASE > NOT > AND > OR
- Phrase matching: intersect docs + check positional adjacency

Term-at-a-Time Query Processing:

- Parse query into AST
- Evaluate AST recursively using set operations
- Apply TF-IDF scoring to matched documents
- Return top-k ranked results

5.4 Resources and References

1. Manning, C.D., Raghavan, P., & Schütze, H. (2008). *Introduction to Information Retrieval.* Cambridge University Press.
2. Salton, G., & Buckley, C. (1988). Term-weighting approaches in automatic text retrieval. *Information Processing & Management*, 24(5), 513–523.
3. Zobel, J., & Moffat, A. (2006). Inverted files for text search engines. *ACM Computing Surveys*, 38(2).
4. Elasticsearch Documentation: `https://www.elastic.co/guide/`
5. NLTK Documentation: `https://www.nltk.org/`
6. Python zlib Documentation: `https://docs.python.org/3/library/zlib.html`