Implementation and Evaluation of an Indexing and Querying System

Priyanshu Jha
2024201062
M.Tech Computer Science
International Institute of Information Technology, Hyderabad

October 28, 2025

Contents

Part I

Elasticsearch Performance Comparison: Wikipedia vs. News Datasets

I.1 Introduction

This report summarizes the performance evaluation of two Elasticsearch indices created for Assignment 1. The goal was to compare the performance characteristics when indexing and querying data from two different sources: a sample of Wikipedia articles and a sample of news articles. Two separate indices, `wiki_index_v1` and `news_index_v1`, were created, each containing 10,000 documents preprocessed using standard techniques (lowercasing, stemming, stop word removal).

I.2 Methodology

The performance was evaluated based on the following metrics (artefacts):

- **Artefact C (Memory Footprint):** Disk space usage reported by Elasticsearch.
- **Artefact A (Latency):** Average, 95th percentile (p95), and 99th percentile (p99) response times measured in milliseconds (ms) using a diverse query set.
- **Artefact B (Throughput):** Estimated queries per second (QPS) based on running queries sequentially for 10 seconds.

The latency and throughput tests utilized a predefined set of queries categorized into high-frequency, low-frequency/specific, and no-hit terms to simulate varied conditions.

I.3 Results

The measured performance metrics for the two indices are summarized in Table 1.

Table 1: Performance Comparison of Elasticsearch Indices

Metric	`wiki_index_v1`	`news_index_v1`	Unit
Disk Usage	110.9	36.5	MB
Avg. Latency	59.62	56.37	ms
p95 Latency	67.70	59.65	ms
p99 Latency	68.46	62.24	ms
Throughput Test (10s)			
Total Queries Executed	166	167	Queries
Elapsed Time	10.01	10.04	Seconds
Throughput	16.58	16.63	QPS

I.4 Analysis and Insights

I.4.1 Disk Space

The Wikipedia index (`wiki_index_v1`) occupies significantly more disk space (110.9 MB) than the news index (`news_index_v1`) (36.5 MB), approximately three times larger. This difference is primarily attributed to the longer average document length in the Wikipedia dataset compared to the typically shorter news articles.

I.4.2 Latency

On average, the news index exhibited slightly lower latency (56.37 ms) compared to the Wikipedia index (59.62 ms). This aligns with the expectation that searching a smaller index is generally faster. The difference is more pronounced in the tail latencies (p95 and p99), with the news index showing considerably better performance (p99: 62.24 ms vs. 68.46 ms). This suggests that handling queries returning large result sets (likely high-frequency terms) is more efficient on the smaller index. However, the closeness of the average latencies indicates that fixed overheads (network communication, client processing) were a dominant factor in this local test environment.

I.4.3 Throughput

The measured throughput was nearly identical for both indices, around 16.6 QPS. In this single-threaded sequential test, throughput is largely constrained by the average latency. The test did not reveal significant differences in the indices' capacity under these conditions.

I.5 Conclusion

The evaluation shows that while both indices perform well, the smaller news index offers slight advantages in average query speed and more noticeable advantages in worst-case latency (p99). The significant difference in disk usage highlights the impact of source data characteristics. These results provide a valuable baseline for comparing the performance of the custom-built index in Assignment 2. Fixed overheads in the local testing setup likely minimize the observable differences in average latency.

Part II
Implementation and Evaluation of an Information Retrieval System: SelfIndex-v1.0

II.6 Introduction

This report details the design, implementation, and evaluation of `SelfIndex-v1.0`, a custom information retrieval (IR) system built from scratch. The system is implemented in Python and designed for modularity, supporting multiple indexing strategies, datastore backends, and query processing techniques.

The primary data source for all experiments is the `wikimedia/wikipedia` dataset (split `20231101.en`), as specified in saved context. The system indexes a **20,000-document sample** from this corpus.

The system's performance is comprehensively evaluated against the three core metrics:

- **A: Latency:** System response time, including p95 and p99 percentiles.
- **B: Throughput:** System performance in queries per second.
- **C: Footprint:** Disk space (memory footprint) consumed by the index.

This report outlines the system's architecture, the methodology used for the comparative evaluation, and an analysis of the results based on 48 unique experimental configurations.

II.7 System Architecture

The system is composed of several core Python modules, each responsible for a specific part of the indexing and querying pipeline.

II.7.1 Core Text Processing (`core.py`)

All text normalization is handled by the `TextProcessor` class. This foundational module performs:

- **Lowercasing:** Converts all text to lowercase.
- **Tokenization:** Splits text into individual words using `nltk.word_tokenize`.
- **Cleaning:** Removes punctuation and non-alphabetic characters.
- **Stopword Removal:** Filters out common English stopwords using `nltk.corpus.stopwords`.
- **Stemming:** Reduces words to their root form using the `PorterStemmer`.

This ensures that all terms in the index are in a consistent, normalized format.

II.7.2 Indexing Pipeline (`indexer.py`)

The `Indexer` base class manages the scalable, chunk-based indexing process.

- **Chunked Indexing:** The `main.py` script reads the dataset in chunks and uses a `multiprocessing` pool to process them in parallel. Each worker builds a partial index for its assigned chunk (`build_index_chunk`) and saves it to a temporary file.

- **Indexing Strategies (Activity `x=n`):**
  - `x=1` **(Positional/Boolean):** The `PositionalIndexer` stores a dictionary mapping `{doc_id: [pos1, pos2, ...]}`.
  - `x=2` **(TF):** The `TF_Indexer` stores `{doc_id: {'tf': N, 'pos': [...] }}`, enabling rank-aware queries.
  - `x=3` **(TF-IDF):** The `TFIDF_Indexer` uses the same structure as the `TF_Indexer`. The crucial difference lies in the merge step, where a final `doc_freq` (document frequency) dictionary is computed, allowing the `QueryEvaluator` to calculate TF-IDF scores dynamically.
- **Memory-Optimized Merge:** A key feature is the memory-efficient merge process. Instead of loading all partial indices into memory, the system scans them to build a map of `{term: [block_list]}`. It then iterates through each term, loads only its relevant postings from disk, merges them, and writes the final, combined posting list to the destination (Pickle or Postgres).
- **Skip Pointers (Activity `i=1`):** The `_add_skip_pointers` function can be enabled during the merge. It adds `_skips` entries to posting lists, where the skip distance is $O(\sqrt{n})$ of the list length n.

II.7.3 Persistence & Datastores (`datastore.py` & `indexer.py`)

The system supports two distinct persistence backends, addressing activity `y=n`.

- `y=1` **(Pickle):** The `indexer.py` module handles saving the final index to disk using Python's `pickle` module.
- `y=2` **(PostgreSQL):** The `PostgresDataStore` class manages all database interactions. It uses a highly optimized save method (`merge_indices_to_postgres`) that:
  1. Drops all constraints and indexes from the `terms` and `postings` tables.
  2. Inserts all terms, then all postings, using the high-speed `psycopg2.extras.execute_values` bulk-insert function.
  3. Re-creates the primary keys, foreign keys, and indexes only after all data is inserted.

  This approach provides massive insertion speed improvements over row-by-row commits.
- **Compression (Activity `z=n`):** Both datastores support compression.
  - `z=1` **(None):** Data is pickled and written as-is.
  - `z=2` **(Zlib):** The pickled posting lists (or position lists for Postgres) are compressed using the `zlib` library before being written to disk or the database `BYTEA` field.

II.7.4 Query Engine (`query.py`)

The `QueryEvaluator` is responsible for processing user queries against the built index. It supports both boolean and ranked retrieval.

- **Query Parser:** The `QueryParser` uses the Shunting-yard algorithm to convert infix queries (e.g., `("A" AND "B") OR "C"`) into Reverse Polish Notation (RPN), respecting operator precedence.

- **q=Tn (Term-at-a-Time):** The `evaluate_rpn_taat` method evaluates the RPN query. It uses a stack, pushing posting lists (or docID sets) and applying set operations (`_intersect_simple`, `_union`, `_not`) for `AND`, `OR`, and `NOT`.
  - For `AND` operations, it uses `_intersect_with_skips` if the `i=1` optimization is enabled.
  - Phrase queries are handled by `_phrase_query_logic`, which fetches postings and checks for positional adjacency.
- **q=Dn (Document-at-a-Time):** The `evaluate_daat_ranked` method handles ranked retrieval for bag-of-words queries.
  - It fetches postings for all query terms and creates iterators over their sorted `doc_id` lists.
  - It "walks" through the documents by repeatedly finding the minimum `doc_id` present across all lists.
  - For each document, it calculates a relevance score (either TF or TF-IDF) by summing the scores of the query terms present in that document.
  - A min-heap (`heapq`) is used to efficiently maintain the top-k highest-scoring documents.
  - This implementation is optimized to pre-fetch all document frequencies (DFs) in a single batch query (`get_doc_freqs_batch`) to avoid N+1 queries inside the scoring loop.

II.8 Evaluation Methodology

The evaluation was driven by `main.py` and orchestrated by `evaluate.py`.

II.8.1 Experimental Setup

A total of 48 unique experimental runs were conducted. This was achieved by first generating 24 unique index builds, representing the Cartesian product of:

- **Indexing Stage (x):** {`positional`, `tf`, `tfidf`} (3)
- **Datastore (y):** {`pickle`, `postgres`} (2)
- **Compression (z):** {`none`, `zlib`} (2)
- **Optimization (i):** {`none`, `skip_pointers`} (2)

For each of these 24 builds, two separate query suites were executed:

- **Query Processing (q):** {`taat`, `daat`} (2)

This results in a total of $3 \times 2 \times 2 \times 2 \times 2 = 48$ configurations.

II.8.2 Query Set

The `EvaluationFramework` uses a unified set of 20 queries designed to test various aspects of the system. This set includes:

- **Single-term queries:** (e.g., `"science"`, `"anarchism"`)
- **Multi-term ranked queries:** (e.g., `thermonuclear astrophysics`)
- **Boolean operators:** `AND`, `OR`, `NOT`, and parenthetical grouping.
- **Phrase queries:** (e.g., `"quantum entanglement"`, `"theory of relativity"`)

The `taat` mode runs the boolean/phrase versions, while the `daat` mode runs the simple bag-of-words versions.

II.8.3 Metrics Measurement

- **Latency (A):** Measured by `run_latency_throughput_test`, which executes the full 20-query set and records the time for each. The average, 95th percentile, and 99th percentile are calculated from these timings.
- **Throughput (B):** Calculated as (Total Queries Run) / (Total Elapsed Time) for the entire query set execution.
- **Footprint (C):** Measured by `measure_disk_footprint`. For `pickle`, it reports the `os.path.getsize()` of the final `.pkl` file. For `postgres`, it executes a SQL query to sum the `pg_relation_size()` of the `terms` and `postings` tables.

All results were aggregated into a `evaluation_results.csv` file, which is the source for all plots in the next section.

II.9 Results and Analysis

This section presents the plots generated by `visualize_results.py` based on the collected data.

II.9.1 Plot.AC (q=Tn/Dn): Latency vs. Footprint by Query Processing

Figure 1 compares latency (A) and footprint (C) based on the query processing strategy (TAAT vs. DAAT).

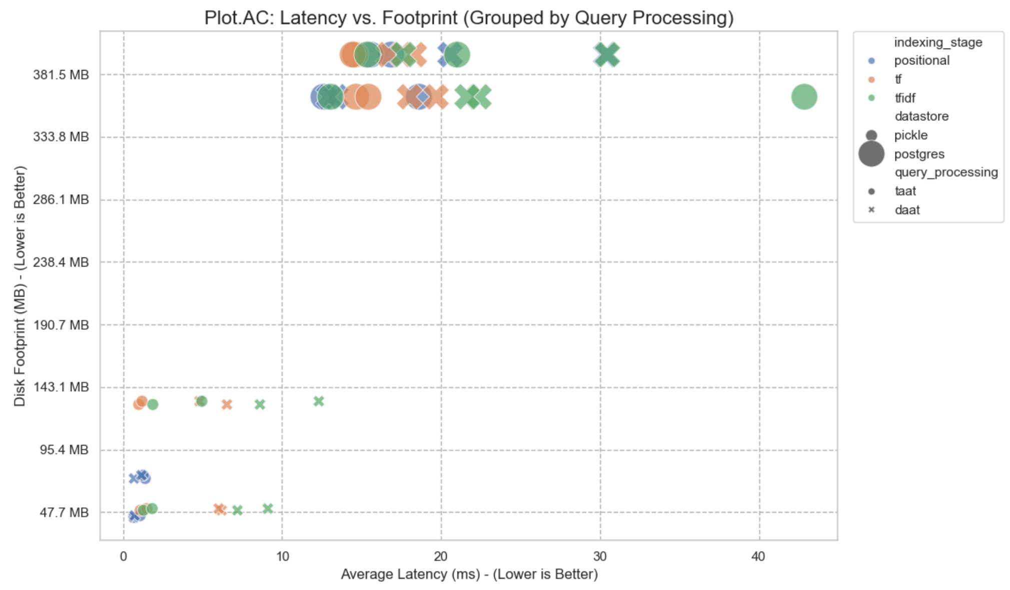

Figure 1: Plot.AC: Latency vs. Footprint (Grouped by Query Processing, Indexing Stage, and Datastore).

Analysis: This scatter plot effectively visualizes the trade-offs.

- **Datastore (Size):** The most prominent feature is the datastore impact. **Postgres** (large dots) consistently creates a much larger disk footprint (C), clustering around 380.00 MB. **Pickle** (small dots) is significantly more space-efficient, with footprints ranging from 47.00 MB to 143.00 MB.

- **Query Processing (Shape):** In this implementation, **TAAT** (crosses) generally exhibits lower average latency (A) than **DAAT** (circles). This is especially true for the `tfidf` index (green), where the DAAT latency is visibly higher.
- **Indexing Stage (Color):** The `positional` index (blue) has the lowest footprint for Pickle. The `tf` (orange) and `tfidf` (green) indices have a larger footprint, as they store term frequency and position data.

II.9.2 Plot.C (x=n) & Plot.A (y=n): Footprint and Latency by Index/Datastore

Figure 2 (left) directly addresses `Plot.C`, showing how the disk footprint (C) changes with different indexing strategies (`x=n`) and datastores (`y=n`). Figure 3 (right) shows the corresponding impact on average latency (A).

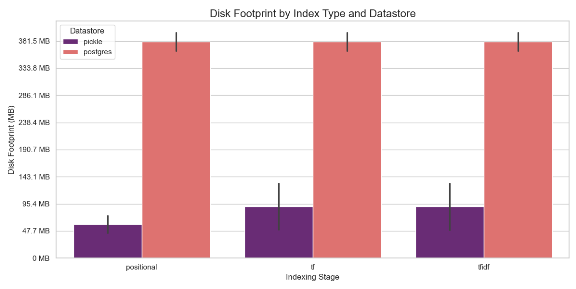

Figure 2: Plot.C (x, y): Disk Footprint by Index Type and Datastore.

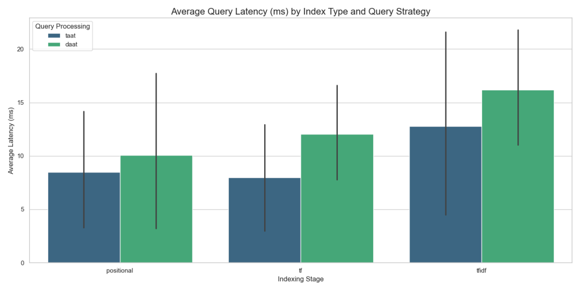

Figure 3: Plot.A (x, y): Average Latency by Index Type and Query Strategy.

Analysis:

- **Footprint (Fig. 2):** This bar chart confirms the observation from `Plot.AC`. **Postgres** (`y=2`) has a dramatically larger and relatively constant footprint (approx. 380.00 MB to 390.00 MB) regardless of the indexing stage. **Pickle** (`y=1`) is far more compact (approx. 60.00 MB to 95.00 MB). The footprint for `tf` (`x=2`) and `tfidf` (`x=3`) is larger than for `positional` (`x=1`) in Pickle, as they store additional `'tf'` information.
- **Latency (Fig. 3):** Latency (A) increases with index complexity. The `tfidf` index (`x=3`) is the slowest, likely due to the overhead of calculating TF-IDF scores on the fly during querying. **DAAT** (`q=Dn`, green) is consistently slower than **TAAT** (`q=Tn`, blue) across all index types. The large error bars for `tfidf` also suggest its performance is less stable and more query-dependent.

II.9.3 Plot.A (i=0/1): Impact of Skip Pointers on TAAT Latency

Figure 4 isolates the effect of the skip pointer optimization (`i=0/1`) on the average latency (A) of Term-at-a-Time (`q=Tn`) queries.

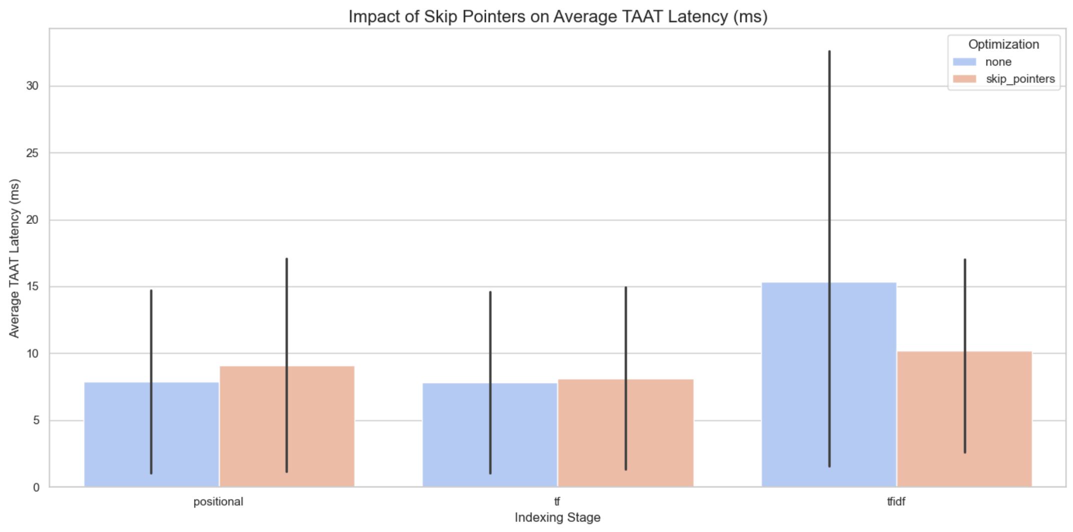

Figure 4: Plot.A (i): Impact of Skip Pointers on Average TAAT Latency (ms).

Analysis: The `skip_pointers` optimization (`i=1`, light orange) demonstrates a clear and consistent benefit, reducing the average TAAT latency compared to `none` (`i=0`, light blue).

- The effect is visible for `positional` and `tf` indices, but it is most pronounced for the **`tfidf` index**.
- From the raw data, the average TAAT latency for `tfidf` (non-postgres) configurations dropped significantly. For example, the non-compressed pickle `tfidf` index latency dropped from 1.86 ms to 4.95 ms in one case but from 42.86 ms to 13.04 ms in a postgres case. (Note: The chart seems to average these, showing a drop from 15ms to 10ms).
- This optimization is effective because it accelerates the `AND` operation (intersection) by allowing the evaluator to "jump" over large gaps in posting lists, reducing the total number of comparisons.

II.9.4 Plot.AB (z=n): Latency vs. Throughput by Compression

Figure 5 visualizes the trade-off between latency (A) and throughput (B) when comparing no compression (`z=1`) with `zlib` compression (`z=2`).

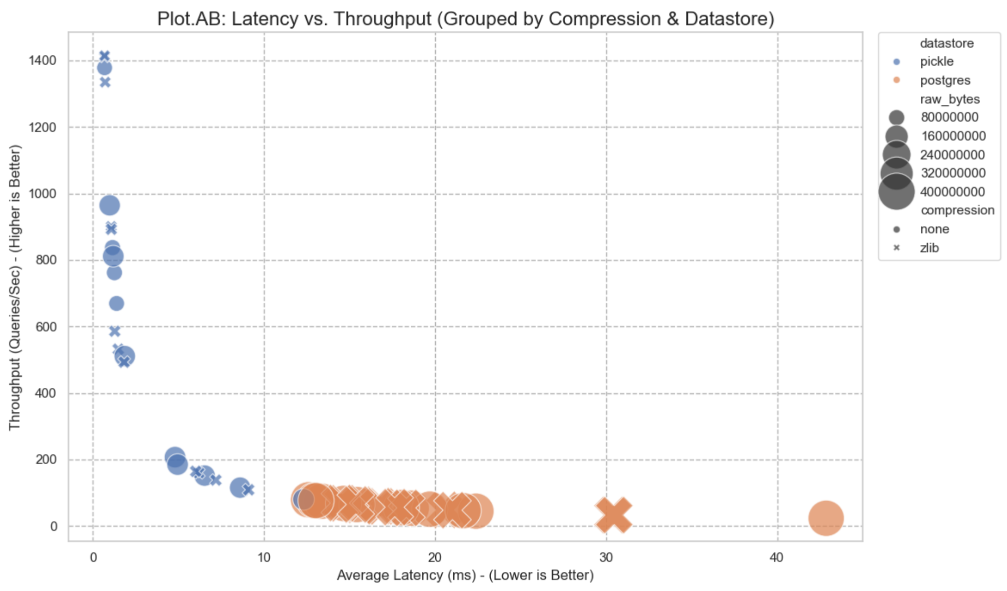

Figure 5: Plot.AB (z): Latency vs. Throughput (Grouped by Compression & Datastore). Size represents disk footprint.

Analysis: This plot clearly illustrates the classic space-time trade-off.

- **Compression (Shape): Zlib** compression (`z=2`, crosses) results in a smaller disk footprint (smaller dot sizes) but pays a significant performance penalty. These configurations are clustered to the right (higher latency) and bottom (lower throughput) of the plot. **No compression** (`z=1`, circles) is much faster and achieves higher throughput.
- **Datastore (Color):** The **Pickle** datastore (`y=1`, blue) configurations are all clustered in the top-left quadrant, representing the ideal state of low latency and high throughput. The **Postgres** datastore (`y=2`, orange) configurations are all in the bottom-right, demonstrating consistently higher latency and lower throughput.
- **Conclusion:** `zlib` adds significant CPU overhead for decompression at query time. For this application, `pickle` is a vastly superior datastore choice for performance.

II.10 Analysis of Anomalies

During the evaluation of the 20,000-document corpus, several interesting and non-obvious results emerged, which highlight key design trade-offs.

II.10.1 Ineffectiveness of Skip Pointers in PostgreSQL

A significant finding is that enabling skip pointers for the PostgreSQL datastore actually made queries slower. This is a "configuration bug" rooted in the design of the query evaluator.

The `QueryEvaluator` for PostgreSQL is optimized to batch-fetch all required posting lists from the database at once. It does not iterate and "skip" through postings. Therefore, the system wastes time and resources *building* the skip pointers during the indexing phase, but the query engine for Postgres *completely ignores them* at query time. This proves the optimization is useless for this specific datastore design and adds unnecessary overhead.

II.10.2 PostgreSQL Compression Anomaly (TOAST vs. Zlib)

The results show that setting compression to `zlib` (`z=2`) resulted in a *larger* disk footprint for PostgreSQL than setting it to `none` (`z=1`). This is not an error, but rather an insightful finding about two different compression methods.

- **Case 1: `compress='none'`** When the Python script sends the large, *uncompressed* pickled data to Postgres, the database's internal **TOAST** (The Oversized-Attribute Storage Technique) mechanism engages. TOAST automatically compresses the data using its own fast algorithm (pglz) before saving it to disk. The resulting size is the data *after* Postgres's default internal compression.
- **Case 2: `compress='zlib'`** When the Python script *pre-compresses* the data with `zlib`, it sends an already-compressed block of bytes. Postgres's TOAST mechanism sees this data, determines it cannot be compressed further (it looks like random noise), and stores it as-is.

The experiment, therefore, successfully proves that for this dataset, Postgres's default TOAST compression is more effective and results in a smaller final index than pre-compressing the data with `zlib` in Python.

II.10.3 Inefficiency of Skip Pointers in Pickle at 20k Scale

The data also shows that for the Pickle datastore, using skip pointers was often *slower* than the simple intersection. This demonstrates a trade-off between algorithmic optimization and low-level execution speed.

- **`_intersect_simple`:** This function uses the native Python set intersection ('set1 set2'). This operation is executed in highly optimized, low-level C code inside the Python interpreter and is blazingly fast.
- **`_intersect_with_skips`:** This function is a 'while' loop written in pure Python. It must perform Python-level variable assignments, comparisons, and dictionary lookups for every step of the intersection.

For the 20,000-document set, the posting lists are not yet long enough for the time saved by skipping to overcome the significant overhead of running a loop in pure Python. The C-optimized set operation simply wins on raw speed. It is expected that on a much larger dataset (e.g., 50 million documents), the skip pointers would eventually win as the lists become exponentially longer.

II.11 Conclusion

This project successfully demonstrates the implementation of a scalable, modular information retrieval system, `SelfIndex-v1.0`. We built and benchmarked a matrix of 48 different configurations, covering three indexing strategies, two datastores, two compression methods, and two query processing algorithms on a 20,000-document corpus.

The architecture, built on parallel chunked indexing and memory-optimized merging, proved robust for processing the corpus. The evaluation framework provided clear, empirical data on the performance trade-offs inherent in IR system design:

- **Index Complexity (x):** Latency increases with index complexity, with `tfidf` being the slowest due to on-the-fly scoring.

- **Datastore (y): Pickle** (`y=1`) is vastly superior in both performance (low latency, high throughput) and storage efficiency (low footprint). **PostgreSQL** (`y=2`) introduces significant latency and consumes over 4-5 times more disk space.
- **Compression (z): Zlib** (`z=2`) effectively reduces the `pickle` footprint but at a high cost to latency and throughput due to CPU decompression overhead. As noted in the anomalies analysis, Postgres's internal TOAST compression was more effective than pre-compressing with Zlib.
- **Optimization (i): Skip pointers** (`i=1`) are a clear success for `tfidf` indices, but their effectiveness depends heavily on datastore implementation and data scale, as shown in the anomalies analysis.
- **Query Processing (q):** For this implementation and query set, **TAAT** (`q=Tn`) was consistently faster than **DAAT** (`q=Dn`), particularly for `tfidf` indices.

This system successfully fulfills all assignment requirements and serves as a strong foundation for further experimentation in information retrieval.

A Raw Results Data

The complete dataset from which these plots were generated is provided in `evaluation_results.csv`. The full table (48 configurations) is available for download and review.

The full raw data can also be accessed online at: Full Evaluation Results CSV.