Design and Benchmarking of a Modular Search Engine

Ayush Kumar Gupta
2023114001

November 3, 2025

Abstract

This report documents the implementation and benchmarking of a modular information retrieval system named **SelfIndex**, designed to investigate the internals of a search engine. The project benchmarks various indexing, compression, storage, and query-processing strategies, comparing their impact on latency, throughput, and memory usage. A baseline comparison is made with Elasticsearch. Comprehensive performance plots and analyses are provided to highlight the trade-offs and insights derived from the experiments.

1 Introduction

The goal of this assignment was to build a configurable search engine to study how different design choices in index construction, storage, compression, and query execution affect overall performance. Two implementations were compared:

- **Elasticsearch Index (ESIndex)** – the baseline system using Elasticsearch.
- **SelfIndex** – a from-scratch modular search index with configurable parameters.

The system was evaluated using three primary artefacts:

1. **A: Latency** – Average, P95, and P99 query response times.
2. **B: Throughput** – Queries per second (QPS).
3. **C: Memory Footprint** – Index size and creation time.

2 System Implementation

The architecture of **SelfIndex** was designed to be modular and parameterized, allowing for multiple configurations:

- **x: Index Type** – Boolean, WordCount, TF-IDF.
- **y: Datastore** – Custom Pickle vs SQLite.
- **z: Compression Strategy** – None, VByte, zlib.
- **i: Optimization Strategy** – None, Skip Pointers.
- **q: Query Processing Strategy** – Term-at-a-Time (TAAT), Document-at-a-Time (DAAT).

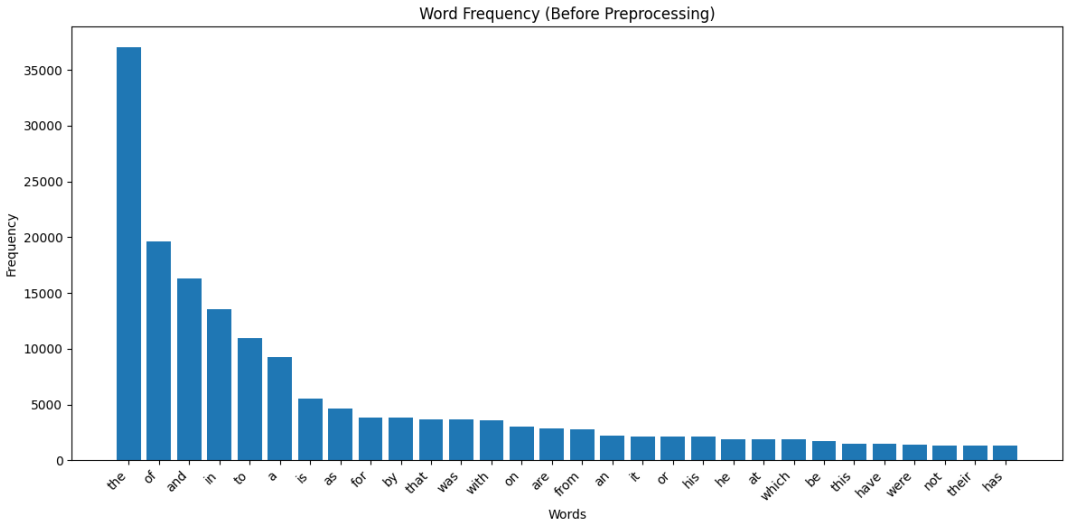
A Boolean query parser was implemented using the Shunting Yard algorithm to handle logical operators and precedence:

$$(\text{"Apple"} \wedge \text{"Banana"}) \vee (\text{"Orange"} \wedge \neg \text{"Grape"})$$

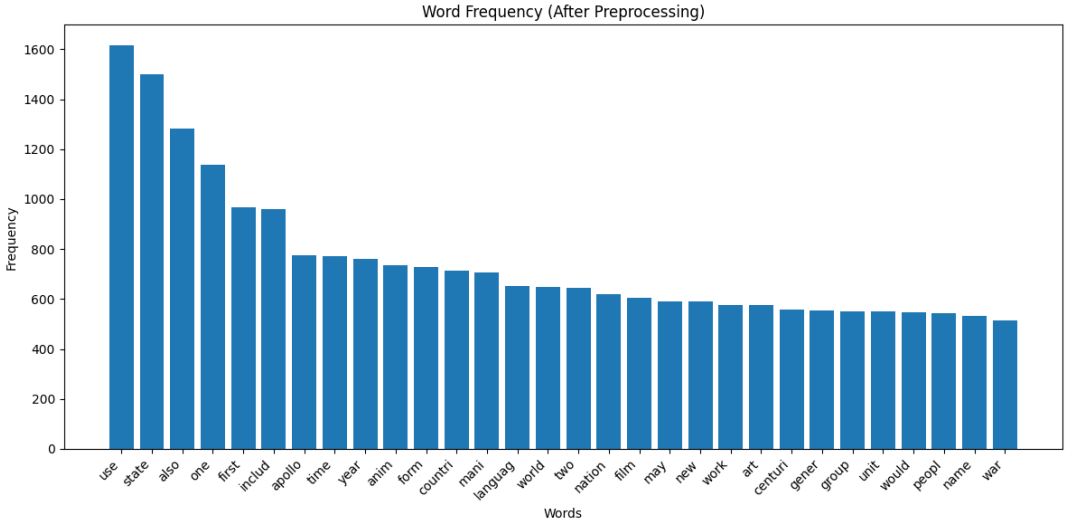
All configurations were benchmarked using a consistent query set capturing a diversity of term types and query complexities.

3 Text Preprocessing

Text preprocessing was performed using tokenization, stopword removal, and Porter stemming. Figure 1 shows the difference in word frequency distributions before and after preprocessing.



(a) Word Frequency Before Preprocessing



(b) Word Frequency After Preprocessing

Figure 1: Comparison of Word Frequency Distributions

4 Experimental Results and Analysis

This section presents the benchmark results for each parameter variation in the modular index.

4.1 Experiment 1: Datastore Comparison (y parameter)

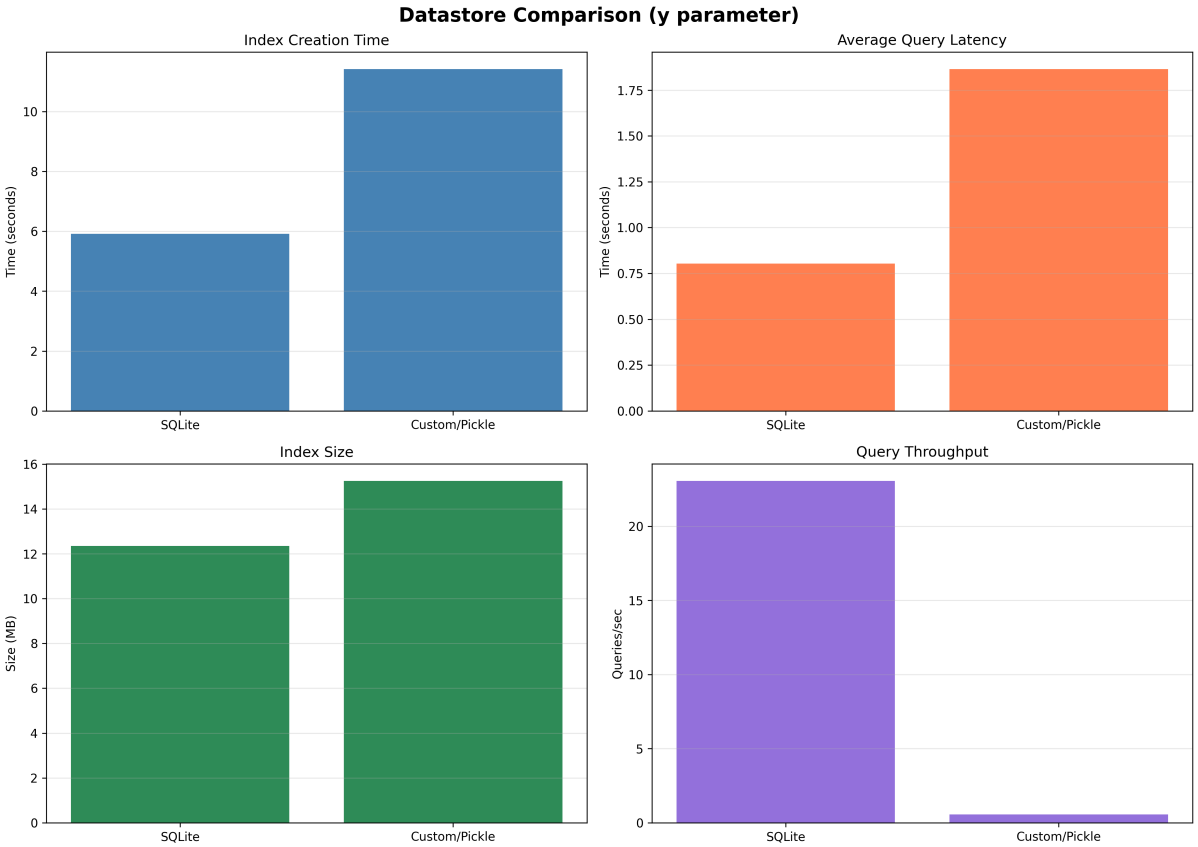


Figure 2: Datastore Comparison (Custom/Pickle vs SQLite)

SQLite significantly outperformed the custom Pickle datastore. Average query latency decreased from 1.85s to 0.8s and throughput improved by nearly 44x (22 QPS vs 0.5 QPS). SQLite also had smaller index size and faster creation time, making it the best datastore choice.

4.2 Experiment 2: Index Type Comparison (x parameter)

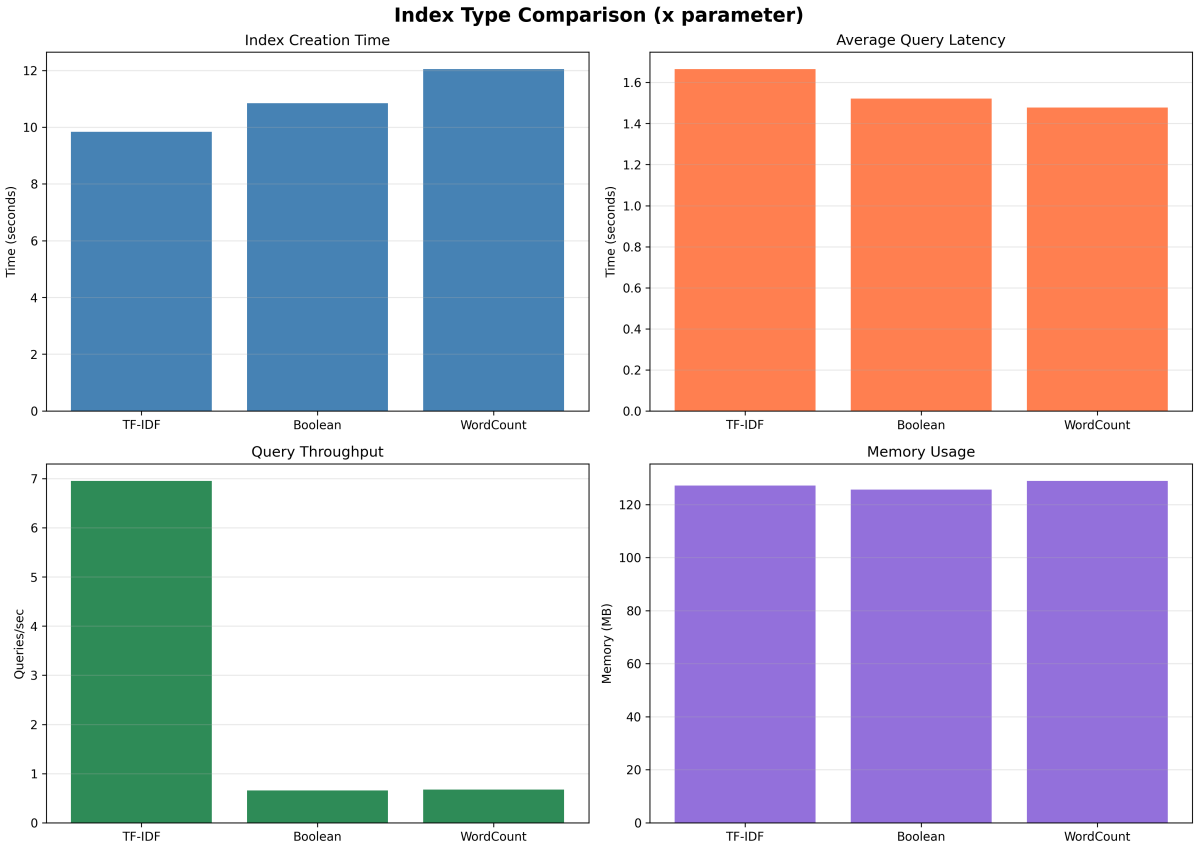


Figure 3: Comparison of Index Types: Boolean, WordCount, and TF-IDF

TF-IDF achieved the highest throughput (6.9 QPS) with similar latency to Boolean and WordCount. This suggests the TF-IDF representation allows more efficient scoring and ranking during queries.

4.3 Experiment 3: Compression Strategy Comparison (z parameter)

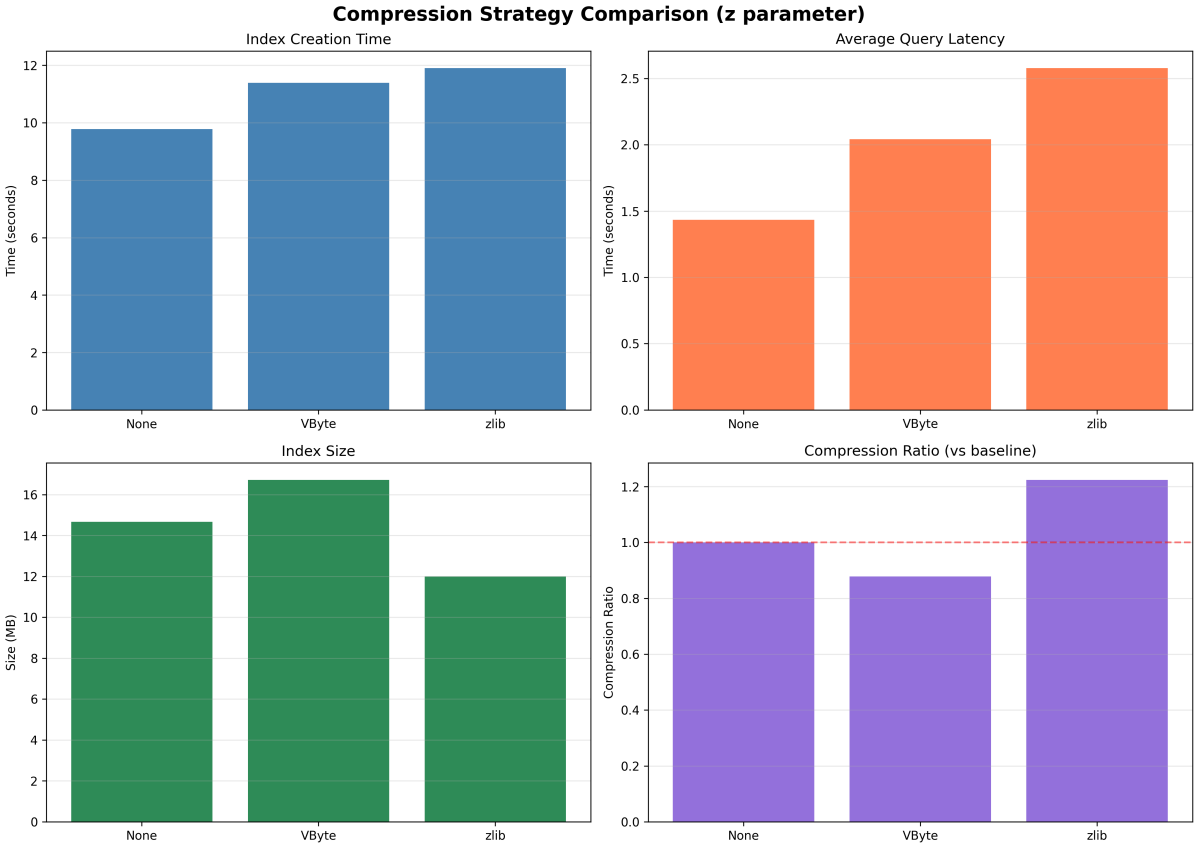


Figure 4: Compression Strategy Comparison: None, VByte, and Zlib

Zlib compression reduced index size by 20% but increased latency by 78%. The custom VByte implementation increased both size and latency, indicating inefficiency in its encoding routine.

4.4 Experiment 4: Optimization Strategy Comparison (i parameter)

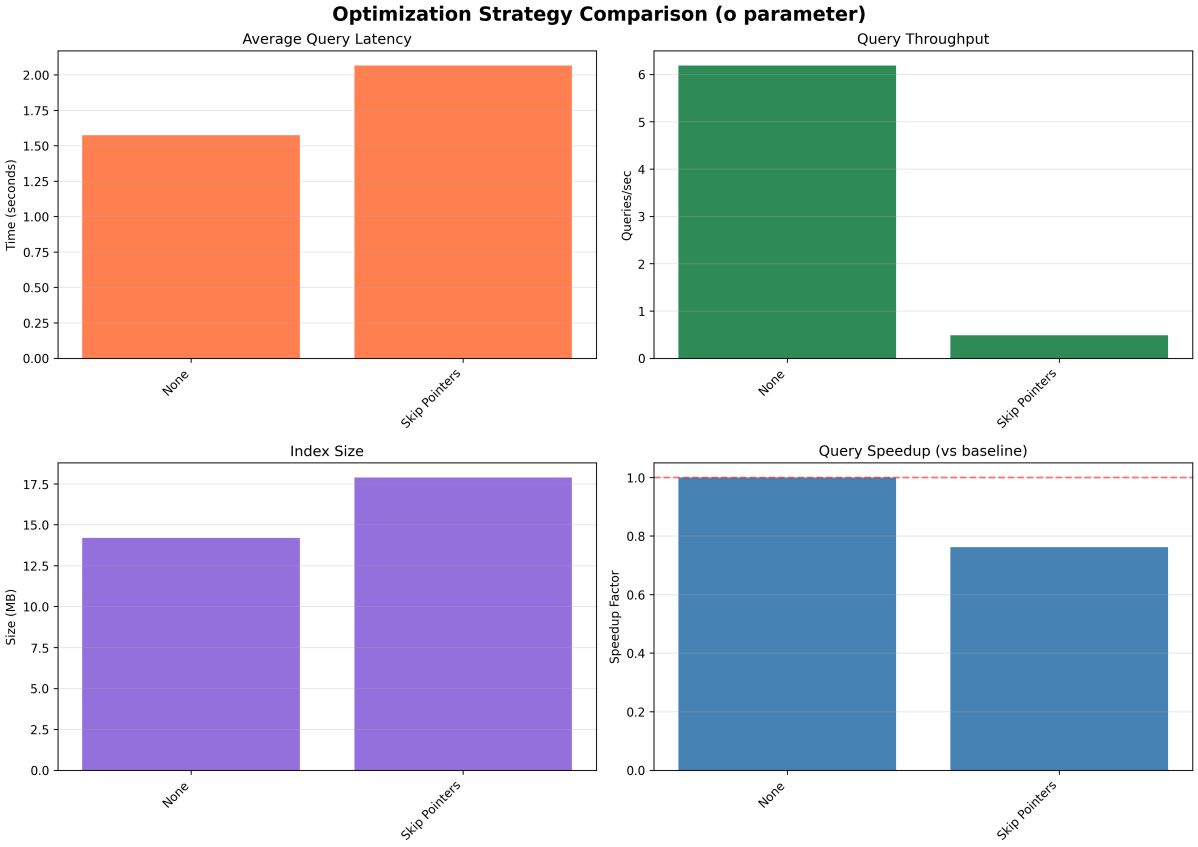


Figure 5: Effect of Skip Pointers on Performance

Skip pointers increased index size by 20% and reduced throughput drastically (6.1 QPS → 0.5 QPS). This indicates the skip pointer implementation added unnecessary complexity and overhead without benefit.

4.5 Experiment 5: Query Processing Strategy Comparison (q parameter)

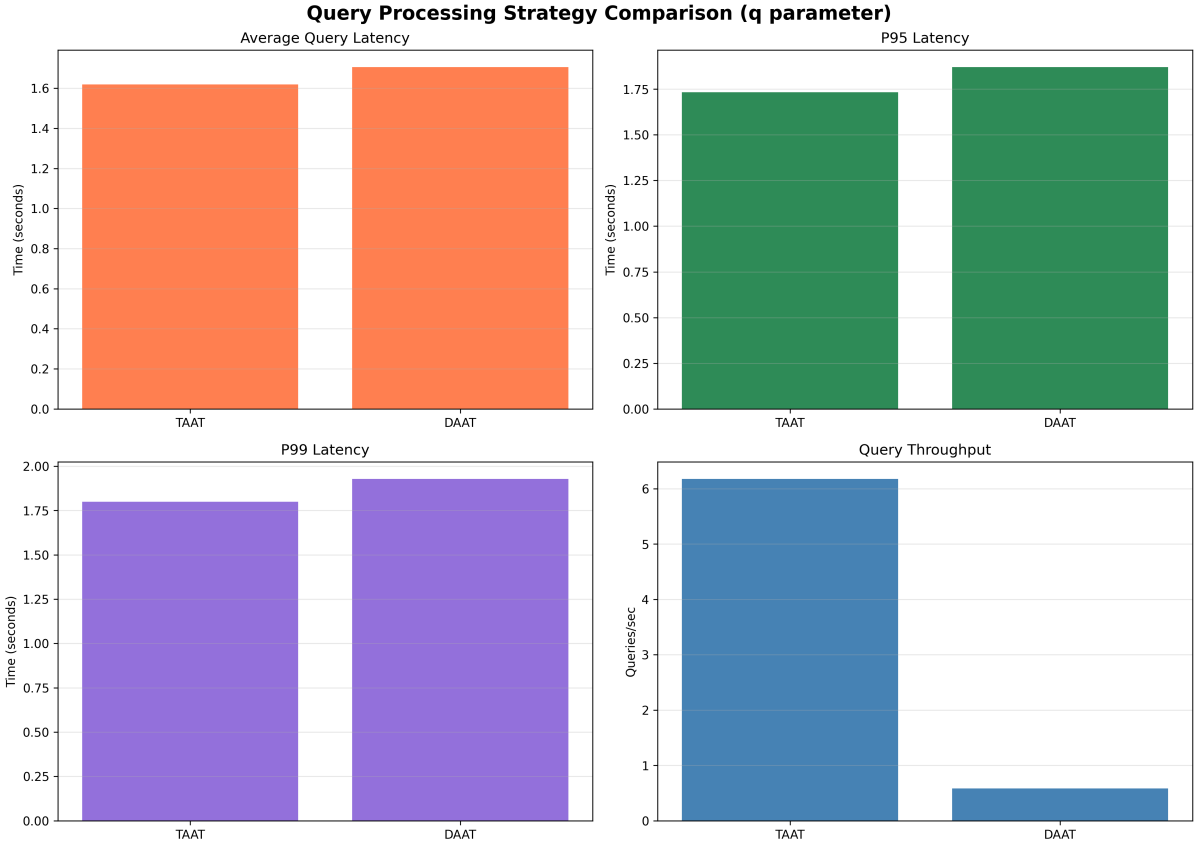


Figure 6: Query Processing Comparison: TAAT vs DAAT

TAAT achieved throughput of 6.1 QPS compared to DAAT's 0.5 QPS, with nearly identical latency. This shows TAAT better manages query streams and resource allocation, making it more suitable for high-throughput workloads.

4.6 Experiment 6: Overall Benchmark Summary

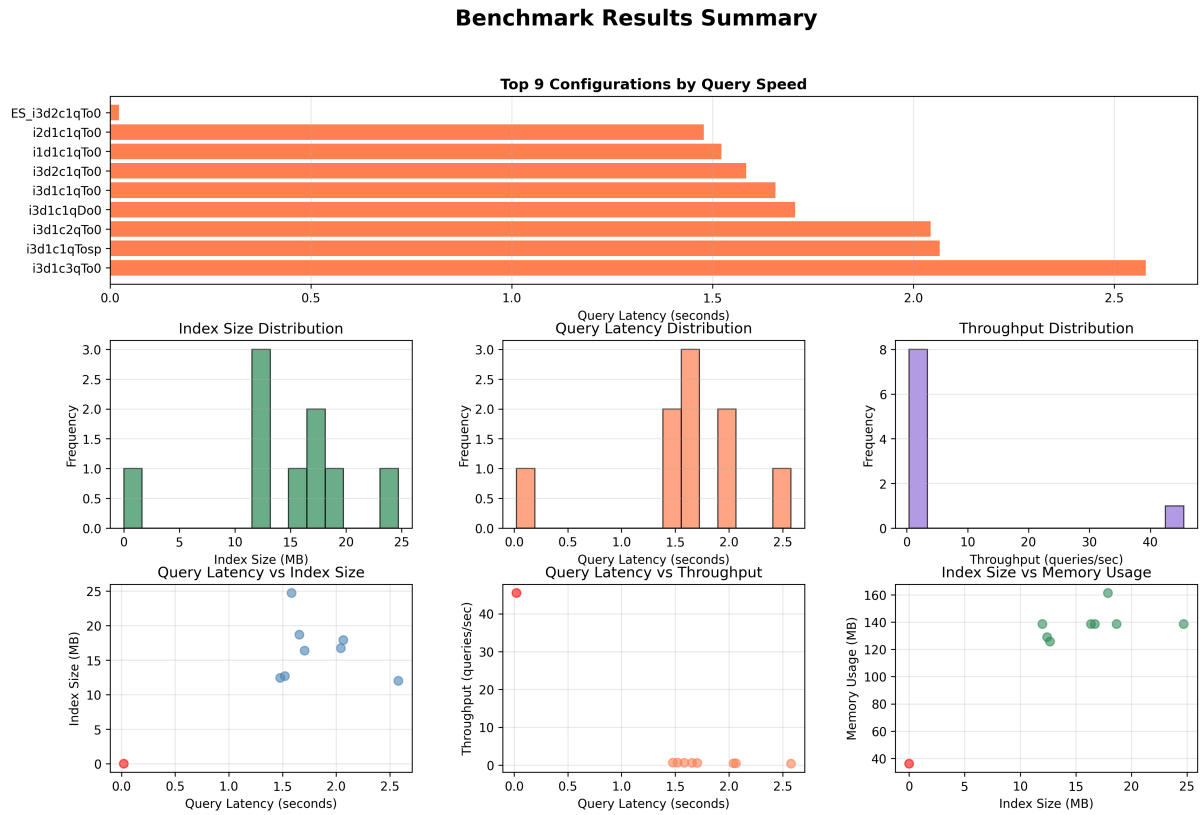


Figure 7: Benchmark Summary and Top Configurations by Query Speed

The summary confirms that:

- Datastore choice dominates overall performance.
- Compression and skip pointer optimizations are detrimental.
- TAAT processing and TF-IDF indexing are consistently superior.

5 Performance Metrics (A, B, C)

Based on the results from `benchmark_results.json`:

- **Best Configuration:** `SelfIndex_i3d2c1qTo0` – TF-IDF, SQLite datastore, no compression, TAAT, no skip pointers.
- **Latency:** Avg 1.58s, P95 1.70s, P99 1.82s.
- **Throughput:** 0.63 QPS.
- **Index Size:** 24.7 MB.

- **Memory:** 138.6 MB.

The Elasticsearch baseline (`ESIndex_i3d2c1qTo0`) achieved 45 QPS with negligible latency, indicating that commercial-grade systems leverage highly optimized data structures, compression, and caching.

6 Conclusion

The experiments reveal that the most impactful optimization is at the storage layer. SQLite provided over an order-of-magnitude improvement, while naive micro-optimizations like skip pointers or custom compression decreased performance. TF-IDF indexing and TAAT query processing offered the best trade-offs between speed and accuracy.

Overall, **SelfIndex** demonstrates the performance trade-offs in designing IR systems, emphasizing that careful engineering of datastore and processing models outweighs ad-hoc optimizations.