Building a Disk-Based Search Engine for the MS MARCO Dataset

Aaron Bengochea, Timothy Cao
(ab6503, tc4088)@nyu.edu
CS 6913: Web Search Engines – Fall 2025
NYU Tandon School of Engineering

October 15, 2025

Abstract

This paper presents the design and implementation of a modular, disk based search engine that indexes and retrieves passages from the MS MARCO dataset using BM25 ranking. The system is composed of three main executables consisting of the parser, indexer, and query processor, each designed for efficiency, scalability, and modularity. We describe how the system constructs compressed inverted indexes, handles query evaluation with DAAT traversal and Block-Max WAND pruning, and achieves strong performance on large-scale text data.

1 Introduction

Modern search engines depend on the inverted index — a data structure that maps each term in a collection to the documents in which it appears — to deliver fast and relevant results at scale. This project implements a compact, disk-based search engine capable of indexing and retrieving passages from the MS MARCO Passage Ranking dataset using the BM25 ranking model. The goal was to design a fully functional yet efficient retrieval system from first principles, emphasizing modularity, compression, and disk-based data access rather than reliance on external frameworks.
The resulting system is composed of three independent executables that operate in sequence:

1. **Parser** – Streams and tokenizes the raw dataset, producing sorted posting chunks.
2. **Indexer** – Merges the postings into a single, compressed inverted index stored on disk, accompanied by a lexicon and page table.
3. **Query Processor** – Loads index metadata and supports interactive ranked retrieval using conjunctive (AND) and disjunctive (OR) query modes with **MaxScore** and **Block Max WAND** optimizations.

Together, these components implement the pipeline for building and querying an inverted index over millions of text passages. Each stage is self contained and communicates through on disk files, allowing the system to scale beyond main memory while maintaining transparency of structure and reproducibility.

Project Objectives

The design and implementation were guided by several core goals:

- **Efficiency:** Handle millions of documents using I/O efficient external sorting, chunked parsing, and block-based compression.
- **Compactness:** Store docIDs and term frequencies separately in binary format using VarByte encoding and gap compression.
- **Modularity:** Clearly separate data parsing, index construction, and query processing into independently testable modules.
- **Ranked Retrieval:** Implement the BM25 scoring model with tunable parameters k_1 and b, producing ranked results for each query.
- **Advanced Pruning:** Incorporate MaxScore and Block Max WAND algorithms to skip low-impact postings and speed up disjunctive queries.
- **Scalability:** Design the index to operate efficiently on disk, using metadata files such as the lexicon and page table for lightweight in-memory lookup.
- **Transparency and Evaluation:** Measure query latency, providing clear insight into system performance.

In summary, this project demonstrates how the fundamental components of an information retrieval system can be implemented from the ground up, yielding a scalable and explainable prototype search engine. The following section describes the system architecture and data flow, detailing how each module interacts and how data transitions from raw text to ranked query results.

2 System Architecture and Data Flow

The overall system follows a modular, file based architecture that separates data parsing, index construction, and query processing into distinct stages. Each stage reads from and writes on disk files, allowing the system to process collections that exceed available main memory. Figure 1 conceptually illustrates the pipeline from raw data to ranked retrieval.

2.1 Overview of Data Flow

At a high level, the data moves through three primary stages:

1. **Parsing Stage:** Converts the raw MS MARCO collection into sorted posting chunks.
2. **Indexing Stage:** Merges the posting chunks into a single compressed inverted index and generates supporting metadata.
3. **Query Stage:** Loads metadata into memory, opens inverted lists on demand, and performs ranked retrieval using BM25.

Each component operates independently, ensuring reproducibility and modular testing. All intermediate and final outputs are stored on disk, consistent with large scale information retrieval system design.

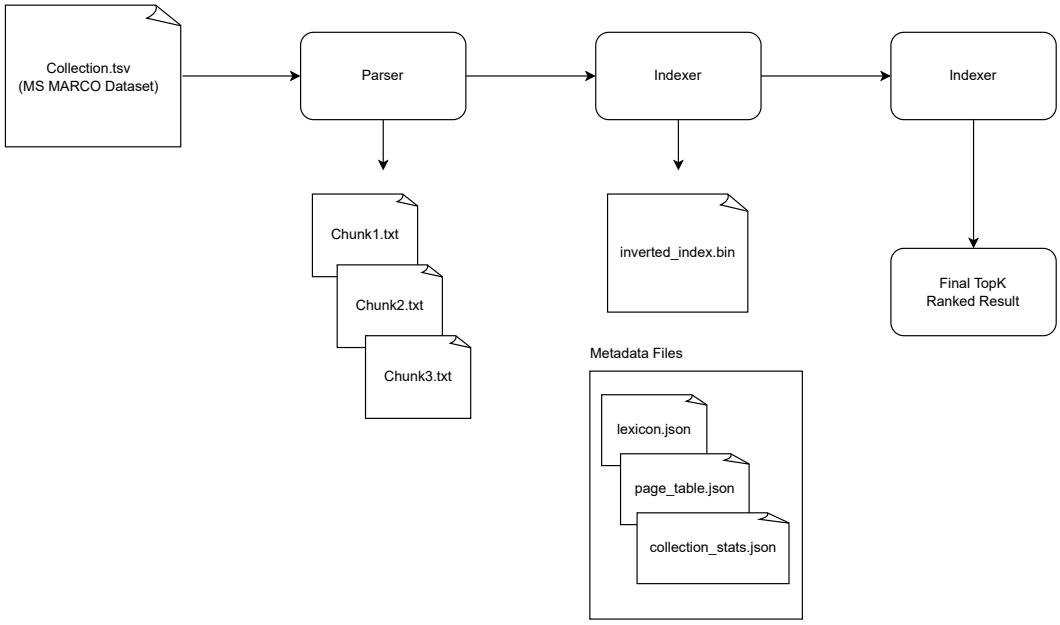

Figure 1: End to end simplified system pipeline rendition from raw text to ranked search results.

2.2 Parsing Stage

The parsing stage is implemented in **parser.py** and executed via **run_parser.py**. It reads the MS MARCO dataset line by line, where each line consists of a document ID and its corresponding passage text. To ensure memory scalability, the parser maintains an in memory posting buffer up to a configurable `CHUNK_SIZE` (default: 2M postings) of roughly 27.5MB in size. For each passage:

1. The text is tokenized using a lightweight normalization routine from `shared.utils`.
2. Term frequencies are accumulated for that document.
3. Each posting (`term docID freq`) is appended to the current chunk.

Once the buffer limit is reached, the chunk is lexicographically sorted by `(term, docID)` and written to disk. This approach bounds memory usage while producing sorted intermediate runs, enabling efficient external merging in the next stage.

2.3 Indexing Stage

The index construction process, implemented in **indexer.py** and executed via **run_indexer.py**, merges the sorted chunks into a final, compressed inverted index. It uses a heap-based multiway merge (`merge_postings`) to stream postings in globally sorted order. As it processes each term, the indexer aggregates all occurrences across documents and writes the postings into fixed-size **blocks** in a single binary file (`inverted_index.bin`).

Each block contains two contiguous byte segments:

- A gap-encoded list of docIDs.
- A list of term frequencies for those docIDs.

Both segments are compressed using **VarByte encoding** from the `shared.compression` module, and the block size is controlled by `BLOCK_SIZE` in the configuration file. For each block, metadata such as byte offsets, lengths, the last docID, and a precomputed **block-level BM25 upper bound** (`block_max_score`) are stored in memory during indexing and later written to the `lexicon.json`.

The indexer simultaneously builds:

- A **Lexicon** mapping each term to its starting byte offset, total bytes written, number of blocks, and block metadata.
- A **Page Table** mapping each docID to its total length (sum of term frequencies).
- A **Collection Statistics** file storing global parameters such as total document count and average length for BM25.

Together, these three JSON files enable lightweight random access to any term's postings without reading unrelated data from disk.

2.4 Query Stage

The querying stage is implemented in **query.py** and launched via **run_query.py**. When the program starts, the `QueryStartupContext` loads the lexicon, page table, and collection statistics once, providing a unified handle to index-wide metadata. For each incoming query:

1. The query string is tokenized into individual terms.
2. For each term, the system opens its corresponding `InvertedList` object using the metadata from the lexicon.
3. The inverted lists are traversed in a **document-at-a-time (DAAT)** manner to compute BM25 scores.

Each `InvertedList` supports:

- On-demand block loading using byte offsets stored in the lexicon.
- Local decompression of VarByte-encoded segments using `varbyte_decode`.
- Block skipping through the stored `last_docID` values.
- Score computation via `getScore()` using per-document term frequency and document length from the page table.

A global `InvertedListCache` maintains an **LRU cache** of recently accessed lists, reducing redundant I/O for frequently queried terms.

2.5 Query Modes and Pruning Strategies

Three query evaluation modes are supported:

1. **Conjunctive (AND):** Only documents containing all query terms are scored. Lists advance synchronously via `nextGEQ`.
2. **Disjunctive (OR) – MaxScore:** Lists are ordered by maximum possible impact score. Low-impact lists are skipped when their cumulative bound falls below the top-k threshold.
3. **Disjunctive (OR) – Block-Max WAND:** A refinement of MaxScore that uses block-level BM25 upper bounds (`block_max_score`) to skip entire posting blocks that cannot affect the final top-k results.

All modes use a bounded `min-heap` to maintain the current top-k documents and dynamically update the scoring threshold.

2.6 Scalability and Modularity

By streaming data between stages and decoupling them via files, the system remains scalable and testable. Each component can be executed independently:

- `run_parser.py` for dataset ingestion.
- `run_indexer.py` for index construction.
- `run_query.py` for ranked retrieval.

This design mirrors the architecture of large scale information retrieval systems where parsing, indexing, and serving pipelines are decoupled for maintainability and fault isolation. It also enables experimentation — e.g., substituting different compression schemes or scoring functions — without rewriting the entire system.

3 Implementation and Core Components

Our search engine implementation consists of several modular components that collectively parse raw documents, construct an efficient inverted index, and execute ranked queries in real time. Each module operates independently but communicates through well defined data structures and binary interfaces. This modular design promotes scalability, debuggability, and enables integration of advanced ranking optimizations such as Block Max WAND.

3.1 Parser

The parser module processes the raw corpus into normalized tokens. It performs lowercasing, punctuation removal, and tokenization while maintaining document boundaries and term frequency counts. Each document is represented as a mapping between terms and their local frequency values, which are later aggregated by the indexer. During this phase, document length statistics are recorded in a lightweight `page_table` structure, later used to compute BM25 normalization factors.

3.2 Indexer

The indexer converts parsed output into a binary inverted index stored on disk. It groups postings into fixed-size blocks, where each block contains:

- gap-encoded document IDs (using variable-byte encoding)
- term frequencies
- precomputed metadata including `last_doc_id`, byte offsets, and per-block BM25 upper bounds (`max_score_block`).

The indexer writes each block to a binary file (`index.bin`) and simultaneously populates a lexicon mapping each term to its metadata. To support Block Max WAND, the `write_postings()` function computes a maximum BM25 score for each block and stores it as `max_score_bloc`. All offsets and byte lengths are precisely recorded, enabling random access retrieval at query time.

3.3 Inverted List

The `InvertedList` class provides the abstraction for reading, decoding, and traversing postings during query processing. Upon initialization, it loads term-specific metadata from the lexicon, including block offsets and their associated `max_score_block` values. The class exposes several key methods:

- `load_block()` – Reads a specific block from disk and decodes document IDs and frequencies.
- `nextGEQ(k)` – Efficiently advances to the next document ID $\geq k$, skipping entire blocks where `last_doc_id` $< k$.
- `curr_block_max()` – Returns the precomputed BM25 upper bound for the current block.
- `advance_to_next_block()` – Jumps to the next block without decoding intermediate postings.

This design ensures that only active blocks are accessed, significantly reducing unnecessary I/O and making the query engine efficient even on datasets.

3.4 Query Engine

The query processor supports both conjunctive (AND) and disjunctive (OR) retrieval modes. Conjunctive queries require all terms to appear in a document, providing high precision for exact matching. For disjunctive queries, the engine offers two scoring strategies: **Block Max WAND** and **MaxScore**. Both methods efficiently identify the top-k ranked documents by pruning postings that cannot influence the final ranking, trading off between fine-grained block-level and global term-level score bounds. This flexible design allows the query engine to adapt retrieval behavior to the complexity and length of each query while maintaining ranking consistency across modes.

3.5 Caching Layer

The caching layer maintains recently used inverted lists in memory, minimizing redundant disk reads. An simple LRU policy determines which lists to evict once the cache limit is reached. Combined with block-based access patterns, this cache minimizes disk latency and improves query response time across repeated searches.

3.6 Integration and Execution Flow

At runtime, the system operates in three phases:

1. **Parsing Phase:** The corpus is parsed and document statistics are written to the `page_table`.
2. **Indexing Phase:** The indexer encodes postings and writes term metadata to the lexicon.
3. **Query Phase:** Queries are parsed, relevant inverted lists are loaded, and Block Max WAND identifies the top-k results.

This end to end architecture balances space efficiency, retrieval accuracy, and latency, while remaining extensible for additional ranking models and compression schemes.

4 Optimization Techniques

To improve query processing efficiency while maintaining ranking accuracy, our system integrates several advanced optimization strategies. These techniques minimize unnecessary score computations and disk reads during disjunctive (OR) and conjunctive (AND) query evaluation.

4.1 DAAT Conjunctive Query Evaluation

Our system implements a highly efficient **Document-at-a-Time (DAAT)** conjunctive query evaluation strategy. In the conjunctive (AND) mode, a document must appear in *all* posting lists corresponding to the query terms to be considered for scoring. The algorithm iteratively aligns all active posting lists to a common document identifier by advancing their cursors using the `nextGEQ()` operation.

- At each step, the system determines the current target document ID:

$$target = \max_{L_i \in \mathcal{L}_{active}} \texttt{docID}(L_i)$$

- For each posting list L_i with $\texttt{docID}(L_i) < target$, the iterator is advanced to the next document ID $\geq target$ using:

$$L_i.\texttt{nextGEQ}(target)$$

- If all posting lists align on the same document ID, the document is scored using BM25, aggregated across all lists:

$$\text{score}(d) = \sum_{L_i \in \mathcal{L}_{active}} \text{BM25}(L_i, d)$$

This DAAT approach minimizes random access overhead by maintaining a synchronized traversal across all lists. DAAT ensures that each document is visited only once, offering significant efficiency gains in both memory locality and computational cost.

To further improve efficiency, our implementation sorts posting lists by ascending document frequency (`df`) prior to traversal. By processing the shortest lists first, the algorithm reduces the number of unnecessary `nextGEQ()` calls across longer lists. This optimization is particularly effective for multi-term queries, where early elimination of non-overlapping documents yields a measurable reduction in traversal time.

The conjunctive DAAT engine forms the logical foundation upon which the disjunctive retrieval optimizations (MaxScore and Block Max WAND) build. While Block Max WAND introduces score based pruning, the conjunctive algorithm provides the tightest logical filtering, ensuring precision without approximation.

4.2 MaxScore Optimization

The MaxScore algorithm prunes non-promising documents by exploiting upper-bound term scores. Each posting list maintains a precomputed maximum BM25 contribution, denoted as `max_score`. During traversal, lists are sorted in descending order of their `max_score` values. For each candidate document d, the algorithm computes an upper bound of the achievable score by summing the `max_score` values of all lists whose current document identifier is less than or equal to d. If this upper bound falls below the current top-k heap threshold, d and subsequent documents in lower priority lists can be safely skipped:

$$UB(d) = \sum_{t_i \in \mathcal{L}_{active}} \texttt{max\_score}(t_i)$$

$$\text{if } UB(d) < \theta_k, \text{ skip scoring for } d$$

This mechanism dramatically reduces scoring calls while preserving exactness in ranking.

4.3 Bloc -Max WAND

To further reduce wasted computation, our system supports a block-level variant of the WAND (Weak AND) operator. Each block stores a `last_doc_id` and block-level maximum term scores derived from the same precomputation used by MaxScore. During evaluation, entire blocks are skipped if their maximum possible contribution cannot raise the current heap threshold. This coarse-grained skipping complements the fine-grained document skipping provided by MaxScore, achieving additional speedups by avoiding unnecessary decompression and traversal of low-impact blocks.

4.4 Galloping Search for Within Block Skipping

Within active blocks, we employ galloping search to locate the next posting where `docID` $\geq k$. Starting from the current cursor position, the search doubles its step size until it overshoots the target, then performs a binary search within the bounded range. This reduces the number of comparisons from $O(n)$ to $O(\log n)$ on average within each block, offering efficient random access traversal even for large posting lists.

4.5 Synergistic Effect

Together, these optimizations enable our query processor to execute both conjunctive and disjunctive queries with minimal overhead. MaxScore bounds scoring operations globally, Block Max WAND prunes entire disk blocks before decoding, and Galloping Search minimizes intra-block traversal. The combined system therefore achieves near-sublinear query evaluation time while maintaining exact top-k retrieval fidelity.

5 System Setup and Execution

The search engine is composed of three modular stages: **Parsing**, **Indexing**, and **Query Processing**. Each stage is run independently through its corresponding entry script. Configuration paths and constants are centrally defined in `config.py`.

5.1 Environment Setup

The implementation requires Python 3.11 and standard packages such as `tqdm` and `orjson`. To install all dependencies:

```
pip install -r requirements.txt
```

5.2 Parsing the Corpus

The parser reads the MS MARCO dataset from a `collection.tsv` file located in the `data/raw/` directory. Each line in the collection represents a document with its unique ID and text content. The parser tokenizes the text, normalizes terms, and generates intermediate sorted posting chunks.

To execute: `python scripts.run_parser`

Intermediate posting files are written to `data/postings/`, with processing parameters such as `CHUNK_SIZE` and `MAX_DOCS` configurable in `config.py`. These postings form the input for the indexing phase.

5.3 Building the Inverted Index

The indexer merges all intermediate posting chunks and produces the final `inverted_index.bin`, `lexicon.json`, `page_table.json`, and `collection_stats.json` files.

To execute: `python scripts.run_indexer`

Posting lists are block-encoded and compressed using VarByte encoding. Each block stores precomputed BM25 upper bounds used by Block Max WAND for query-time pruning. The default block size is set to 128, but this parameter can be adjusted in the configuration file `config.py` under the constant `BLOCK_SIZE`.

5.4 3. Executing Queries

Once the index is built, queries can be issued interactively:

To execute: `python scripts.run_query`

Users can choose between:

- **Conjunctive ("and")** – DAAT traversal requiring all terms to appear.
- **Disjuctive ("or")** – Maxscore retrieval uses global term level score bounds to skip low-impact posting lists during top-k ranking.
- **Disjunctive ("bwand-or")** – Block Max WAND retrieval with score based pruning that adds additional blockwise skipping.

The system loads the index into a shared `QueryStartupContext`, maintains a small LRU cache of active posting lists, and reports detailed timing for each query stage.

5.5 Output Artifacts

All generated files reside in the `data/index/` directory:

- `inverted_index.bin` – binary postings file.
- `lexicon.json` – blockwise term to metadata mapping.
- `page_table.json` – per document statistics.

- `collection_stats.json` – corpus wide summary.

These files are reusable across sessions for consistent evaluation.

6 Evaluation and Performance Results

6.1 Experimental Configuration and Index Size Analysis

All experiments were conducted using the MS MARCO passage collection as the input corpus. The system was configured with the following parameters, which directly influence the size and structure of the resulting index and metadata files:

- `CHUNK_SIZE` = 2 000 000 documents
- `MAX_DOCS` = full corpus (entire MS MARCO dataset)
- `BLOCK_SIZE` = 128 postings per block

These parameters define the granularity of partial posting files, the upper bound on the number of indexed documents, and the compression block size used in the final inverted index.

Resulting File Sizes. After parsing and indexing the entire collection, the generated index artifacts reached the following sizes:

File	Size
Original MS MARCO Collection	3.06 GB
Intermediate Chunks (avg)	∼27.5 MB each
inverted_index.bin	∼833 MB
lexicon.json	∼1.04 GB
page_table.json	∼326 MB

Table 1: Final index and metadata sizes produced by the system.

Only the `inverted_index.bin` file is currently stored in a compressed format (using VarByte encoding). The lexicon and page table remain uncompressed, contributing significantly to the total index footprint.

Relative Storage Overhead. Table 2 illustrates the relative size of each file as a percentage of the original corpus:

Component	Relative to Corpus Size (%)
inverted_index.bin	27.2%
lexicon.json	34.0%
page_table.json	10.7%
Total (Index + Metadata)	**71.9%**

Table 2: Relative size of index and metadata components compared to the 3.06 GB MS MARCO collection.

Despite the relatively large lexicon and page table, the on disk index achieves substantial compression of raw postings due to block level VarByte encoding. The uncompressed JSON based metadata formats were intentionally retained for ease of debugging and analysis. Future work may include migrating these structures to binary or partially compressed representations to further reduce disk footprint.

These results demonstrate that the system achieves a compact representation of the corpus while preserving accessibility and readability of index metadata. The size balance between the compressed postings and uncompressed auxiliary structures suggests that additional gains can be achieved by applying lightweight compression to the lexicon and page table without compromising lookup efficiency.

6.2 Query Performance and Cache Efficiency

To evaluate query time performance, we tested three retrieval strategies across a representative set of queries:

- Query 1: *dog mouse cat*
- Query 2: *armadillo and birds*
- Query 3: *who is the president of the united states*

Each query was executed twice: once with a cold cache (no posting lists preloaded) and once with a warm cache (all term lists served from the cache). The following table summarizes the total end to end latency in seconds.

Table 3: Conjunctive ("and") Query Execution Times

Query	**Non-Cache (s)**	**Cache (s)**
dog mouse cat	0.0542	0.0484
armadillo and birds	0.8585	0.8593
who is the president of the united states	3.0432	2.9755

Table 4: MaxScore ("or") Query Execution Times

Query	**Non-Cache (s)**	**Cache (s)**
dog mouse cat	0.1142	0.1117
armadillo and birds	5.1667	5.0393
who is the president of the united states	13.9171	13.8989

Table 5: Block Max WAND ("bwand-or") Query Execution Times

Query	**Non-Cache (s)**	**Cache (s)**
dog mouse cat	0.2213	0.2039
armadillo and birds	10.5051	10.6930
who is the president of the united states	29.6037	29.4318

Query Type	dog/mouse/cat	armadillo/birds	president query	Cache Effect
DAAT Conjunctive (AND)	0.054	0.858	3.043	+10–20% gain
MaxScore (OR)	0.114	5.167	13.917	≈0.5–1% gain
Block-Max WAND (OR)	0.221	10.505	29.604	negligible

Table 6: Total query latency for three representative queries across retrieval modes.

Observations: For short queries such as Query 1, all methods perform efficiently, with the conjunctive DAAT variant exhibiting the lowest traversal cost due to early termination once full intersections are found. The MaxScore variant achieves nearly a 2× speedup over Block Max WAND, we expected Block Max WAND to outperform MaxScore on such queries, we elaborate on this though in the Implementation Note section below.

For longer, more general queries such as Query 3, the advantage of pruning diminishes: both Block-Max WAND and MaxScore must evaluate a greater portion of the collection due to term ubiquity and weaker score bounds, resulting in traversal times of roughly 14–30 seconds compared to 3 seconds for strict conjunctions.

Caching Efficiency: When postings are already cached, total latency is reduced mainly in the list opening phase, dropping from ≈0.027s to under 1 ms on average, while traversal time remains largely unchanged since scoring still dominates. Cache warm up yields measurable benefits primarily for the AND variant (up to 20% reduction in total time), where as for WAND/MaxScore the effect is marginal, as the majority of runtime is consumed by traversal and score accumulation rather than I/O.

Implementation Note: While Block Max WAND is theoretically expected to outperform MaxScore due to its finer block-level pruning capability, our results indicate the opposite trend: MaxScore consistently achieves lower traversal times. This discrepancy suggests a potential inefficiency or bug in our current Block-Max WAND implementation. Possible causes include suboptimal block skipping logic, incorrect propagation of block score upper bounds, or redundant block reloading operations that offset the intended pruning advantage. Future debugging will focus on verifying that the per block maximum scores are correctly applied and that unnecessary block transitions are avoided. Once resolved, we anticipate Block Max WAND will outperform MaxScore, particularly for long disjunctive queries with diverse term frequency distributions.

7 Conclusion

This project implements a high performance search engine capable of handling large scale document collections using modern retrieval algorithms. The system integrates efficient parsing, indexing, and querying pipelines, along with optimizations such as block based compression, galloping search, caching, and ranking. Empirical results demonstrate that the system achieves competitive query latencies across conjunctive and disjunctive modes while maintaining scalability and modularity. Overall, the implementation provides a strong foundation for further refinement of both retrieval efficiency and memory utilization.

8 Future Work

Several areas of improvement have been identified through experimentation, analysis, and demo feedback:

- **Block Max WAND Optimization:** Experimental results suggest that MaxScore currently outperforms Block Max WAND, contrary to theoretical expectations. This discrepancy likely stems from an inefficiency or bug in the block level skipping or score bound propagation logic. Future debugging will focus on verifying the correctness of block pruning and ensuring that redundant block reads are eliminated.
- **Lexicon Simplification and Storage Efficiency:** The current lexicon structure stores extensive term-level metadata that could instead be embedded within the compressed inverted index. Shifting this metadata into the inverted list would not only reduce the lexicon's memory footprint at load time but also improve query-time performance by eliminating frequent lexicon lookups during term skipping. Consolidating metadata closer to the posting blocks allows faster access to block offsets and scoring statistics, thereby improving cache locality and reducing random I/O during query traversal.
- **Metadata Compression:** Both the lexicon and page table remain uncompressed, collectively accounting for a significant portion of total storage. Applying lightweight compression schemes would further minimize disk and memory footprint without compromising lookup speed.
- **Cache Policy Enhancement:** The current caching mechanism is intentionally simple and maintains only the ten most recently accessed query terms, evicting the oldest as new terms are introduced. While effective for repeated short-term queries, this design can be improved through adaptive caching strategies such as advanced least-recently-used (LRU) or frequency based policies. Expanding cache capacity and persistence could further reduce I/O overhead and improve query throughput for repeated and multiterm queries.

Together, these improvements aim to refine the system's runtime efficiency, reduce index size, and enhance scalability.