MS MARCO SEARCH ENGINE

Assignment #2 - CS 6913 Web Search Engines

NYU Tandon School of Engineering

Prepared By

Ishan Yadav & Riyam Patel

October 16, 2025

Contents

1 Introduction

This report describes the design and implementation of a complete search engine system for the MS MARCO passage-ranking dataset. The system builds an inverted index from 8.8 million text passages and provides ranked search results using the BM25 ranking function.

The system is divided into three main components:

- An **indexer** that parses passages and creates intermediate sorted index blocks
- A **merger** that combines blocks into a compressed final index
- A **query processor** that retrieves and ranks relevant passages

The implementation is written in Python and emphasizes memory efficiency, I/O optimization, and compression techniques suitable for large-scale text retrieval.

2 System Overview

2.1 Architecture

The system follows a three-stage pipeline architecture:

Stage 1 (Indexer): Reads the MS MARCO collection and builds intermediate index blocks using the Blocked Sort-Based Indexing (BSBI) algorithm. Each block contains sorted postings that fit in memory.

Stage 2 (Merger): Performs a k-way merge of all intermediate blocks to create a final compressed inverted index. Uses varbyte encoding and delta-encoding for compression.

Stage 3 (Query Processor): Loads the lexicon and document table into memory, then processes user queries by fetching compressed inverted lists from disk and scoring documents using BM25.

2.2 Data Flow

The system processes data through a sequential pipeline where each stage writes output files to disk that are consumed by the next stage.

The indexer reads the MS MARCO collection, tokenizes each passage, and builds sorted blocks of postings in memory. When a block reaches capacity (5 million postings), it is written to disk and the memory is reused. This produces 50-100 intermediate block files along with document metadata.

The merger performs a k-way merge of all blocks using a min-heap, compressing postings with delta-encoding and varbyte compression. The result is a compact binary index consisting of three files: the lexicon (term $\rightarrow$ offset mappings), inverted lists (compressed postings), and document table (doc_id $\rightarrow$ metadata).

The query processor loads the lexicon and document table into memory at startup, then processes queries by fetching only the relevant inverted lists from disk. Documents are scored using BM25 in a document-at-a-time fashion, with results maintained in a top-k heap.

3 How to Run the System

3.1 Prerequisites

- Python 3.7 or higher
- Required packages: `tqdm` (optional, for progress bars)
- MS MARCO passage collection file (`collection.tsv` format)

3.2 Step-by-Step Execution

Step 1: Build the inverted index (indexer)

```
python indexer.py data/collection.tsv index/blocks --max-postings
    5000000
```

This creates intermediate block files and document metadata in the output directory.

Step 2: Merge blocks into final index (merger)

```
python merger.py index/blocks index/final_index
```

This produces three binary files: `lexicon.bin`, `inverted_lists.bin`, and `doc_table.bin`.

Step 3: Run queries (query processor)

```
python query_processor.py index/final_index data/collection.tsv
```

The system will start an interactive command-line interface where you can enter search queries, choose the number of results (k), and select query mode (conjunctive or disjunctive).

4 Internal Design and Algorithms

4.1 Indexer (indexer.py)

4.1.1 Text Processing

The `PassageParser` class reads the MS MARCO TSV file line by line. Each line contains:

`passage_id [TAB] passage_text`

The parser handles encoding errors gracefully (UTF-8 with `error=ignore`) to accommodate non-English text and special characters.

4.1.2 Tokenization

The `Tokenizer` class applies the following rules:

- Convert text to lowercase
- Extract alphanumeric sequences using regex pattern `[a-z0-9]+`
- Filter tokens to length 2-20 characters
- No stemming or stopword removal (to maintain generality)

This produces consistent tokens for both indexing and query processing.

4.1.3 Blocked Sort-Based Indexing (BSBI)

The `BlockedIndexer` class implements the BSBI algorithm to handle datasets larger than available memory.

Memory Management:

- Preallocates a fixed-size array for postings (default: 5 million postings)
- Each posting is a tuple: (term, doc_id, frequency)
- When the array fills, the block is written to disk and the array is reused

Block Writing Process:

1. Sort postings by term (lexicographically), then by doc_id

2. Group postings by term
3. Write to disk in format: `term [TAB] doc_id:freq,doc_id:freq,...`
4. Reset array for next block

Document Metadata: For each document, the system stores internal doc_id (sequential integers starting from 0), original passage_id (from MS MARCO), and document length (number of tokens). This metadata is written to `doc_metadata.json` for use by the merger.

4.2 Merger (merger.py)

4.2.1 K-Way Merge Algorithm

The `KWayMerger` class merges multiple sorted block files efficiently using a min-heap data structure.

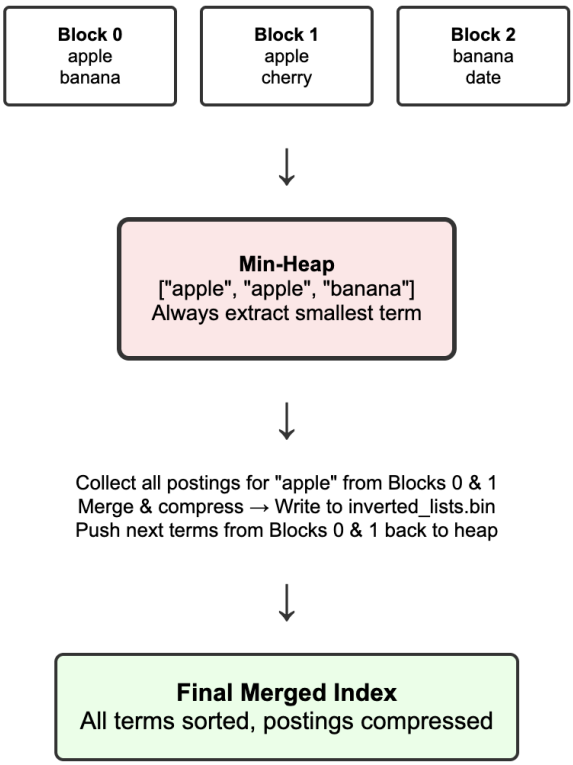

Figure 1: K-way merge algorithm with min-heap

Algorithm 1 K-Way Merge Algorithm

```
Open a BlockReader for each intermediate block file
Initialize a min-heap with the first term from each reader
while heap is not empty do
    Extract the minimum term from the heap
    Collect all postings for that term from all readers
    Merge and sort postings by doc_id
    Write to compressed inverted list
    Push next terms from readers back onto the heap
end while
```

The heap ensures $O(\log k)$ time complexity per term, where k is the number of blocks.

4.2.2 Compression

The system uses two compression techniques that work together to achieve 60-80% compression on posting lists.

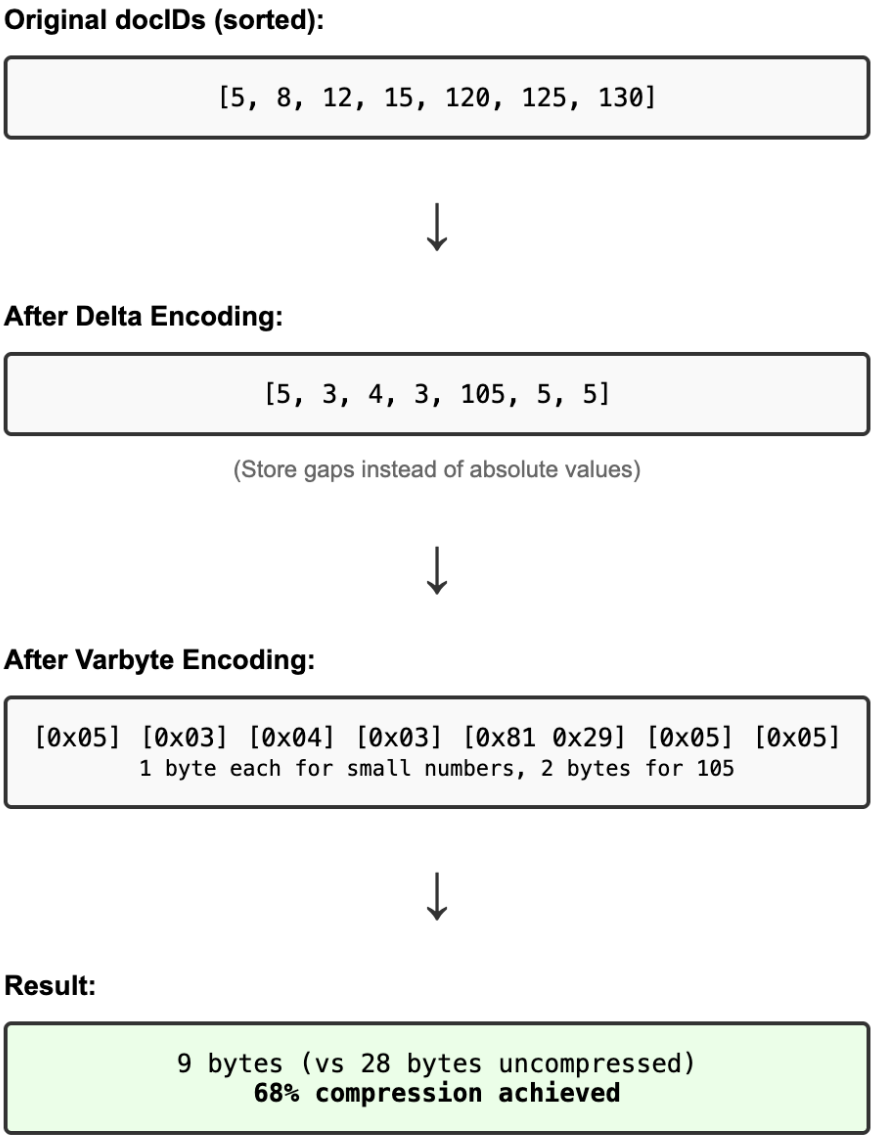

Figure 2: Delta encoding combined with varbyte compression

Varbyte Encoding:

- Variable-length integer encoding using 7 bits for data, 1 bit for continuation
- Numbers < 128 use 1 byte, larger numbers use multiple bytes
- Achieves significant compression for small integers (common in frequency values)

Delta Encoding (for docIDs):

- Stores differences between consecutive doc_ids instead of absolute values
- Example: [5, 8, 12, 15] $\rightarrow$ [5, 3, 4, 3]

- Exploits the fact that doc_ids are sorted and often clustered
- Combined with varbyte, this compresses docID lists by 60-80%

4.2.3 Inverted List Format

Each inverted list is stored in a structured binary format:

```
[Header: 12 bytes]
  - doc_count (4 bytes)
  - docid_length (4 bytes)
  - freq_length (4 bytes)
[Compressed delta-encoded docIDs]
[Compressed frequencies]
```

DocIDs and frequencies are stored in separate blocks (not interleaved) to enable selective decompression, improve cache locality during DAAT traversal, and support skipping operations more efficiently.

4.2.4 Lexicon Structure

The `Lexicon` class maintains an in-memory dictionary mapping each term to:

- **offset:** byte position in `inverted_lists.bin`
- **length:** size of the inverted list in bytes
- **doc_count:** number of documents containing the term

The entire lexicon is loaded into memory at query time for fast term lookup.

4.2.5 Document Table Structure

The `DocumentTable` class stores mappings: doc_id → passage_id (for result display) and doc_id → document length (for BM25 scoring). The document table is fully loaded into memory and provides $O(1)$ lookups.

4.3 Query Processor (query_processor.py)

4.3.1 Inverted List API Framework

The `InvertedListReader` class implements the API framework discussed in class:

- **open(term, lexicon) → bool:** Looks up term in lexicon, seeks to byte offset, reads and decompresses the full inverted list
- **next_doc() → (doc_id, frequency):** Advances to next posting in the list, $O(1)$ operation
- **seek(target_doc_id) → (doc_id, frequency):** Uses binary search to find first doc_id $\geq$ target, $O(\log n)$ operation
- **get_doc_id() → int:** Returns current document ID
- **get_frequency() → int:** Returns term frequency in current document
- **close():** Releases resources and closes file handle

This abstraction hides compression details from query processing algorithms.

4.3.2 BM25 Ranking Function

The `BM25Scorer` class implements the Okapi BM25 formula:

$$\text{score}(D, Q) = \sum_{q_i \in Q} \text{IDF}(q_i) \times \frac{f(q_i, D) \times (k_1 + 1)}{f(q_i, D) + k_1 \times (1 - b + b \times |D|/\text{avgdl})} \tag{1}$$

Where:

$$\begin{aligned}
\text{IDF}(q_i) &= \log\left(\frac{N - \text{df}(q_i) + 0.5}{\text{df}(q_i) + 0.5}\right) \\
f(q_i, D) &= \text{term frequency of } q_i \text{ in document } D \\
|D| &= \text{length of document } D \\
\text{avgdl} &= \text{average document length in collection} \\
N &= \text{total number of documents} \\
\text{df}(q_i) &= \text{number of documents containing } q_i \\
k_1 &= 1.2 \text{ (term frequency saturation parameter)} \\
b &= 0.75 \text{ (length normalization parameter)}
\end{aligned}$$

The scorer caches IDF values to avoid recomputation across documents.

4.3.3 Document-at-a-Time (DAAT) Processing

The system implements both disjunctive (OR) and conjunctive (AND) query processing using document-at-a-time traversal.

Disjunctive Query (OR semantics):

Algorithm 2 Disjunctive DAAT Query Processing

```
Open inverted lists for all query terms
Initialize min-heap with first doc_id from each list
while heap is not empty do
    Extract smallest doc_id
    Collect frequencies from all lists at this doc_id
    Score document using BM25
    Update top-k heap if score is high enough
    Advance all lists that matched this doc_id
end while
Return top-k results sorted by score
```

Time Complexity: $O(L \log k + k \log k)$ where L is total postings processed.

Conjunctive Query (AND semantics):

Time Complexity: $O(L_{\text{shortest}} \times k_{\text{terms}} \times \log L_{\text{avg}})$ where L_{shortest} is length of shortest list.

The conjunctive algorithm is optimized by starting with the shortest list (fewest candidates), using binary search (seek) instead of linear traversal, and early termination when a term is not found.

4.3.4 Top-K Result Maintenance

Both algorithms maintain a min-heap of size k:

- The root contains the lowest-scoring document in the top-k
- New documents are only added if they score higher than the root

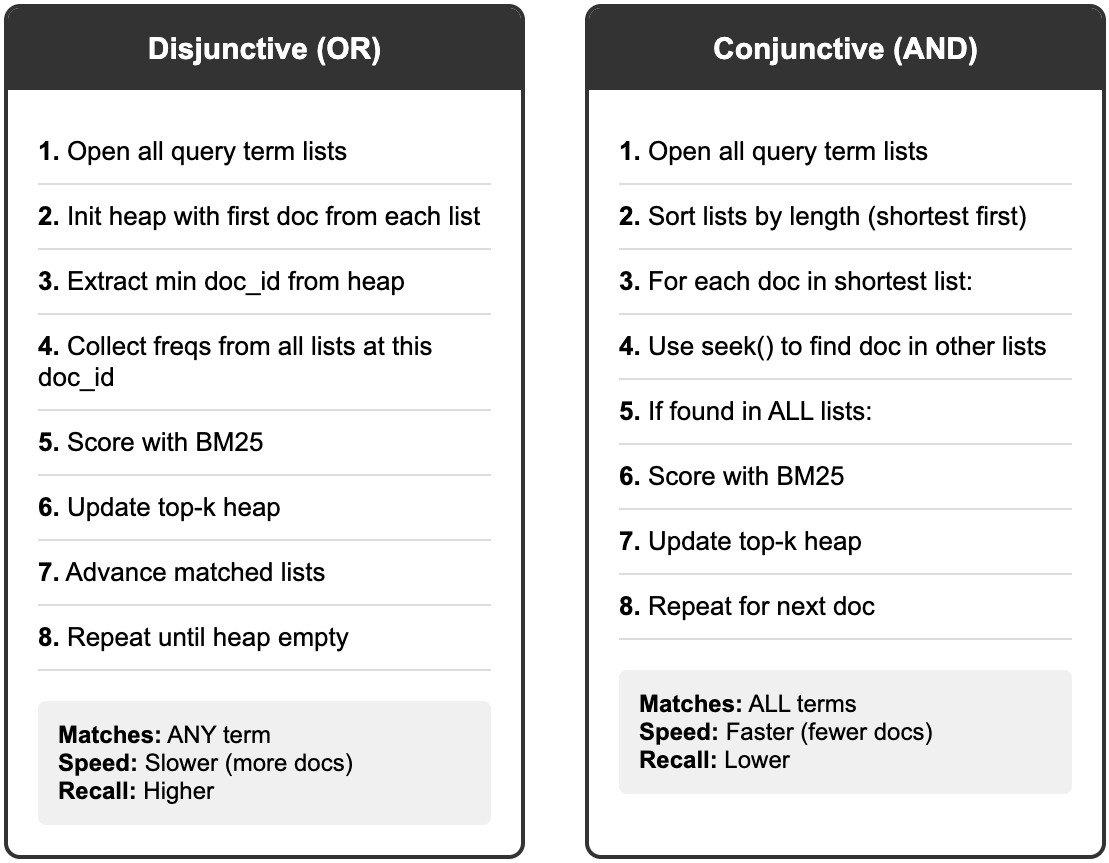

Figure 3: Comparison of disjunctive and conjunctive DAAT algorithms

Algorithm 3 Conjunctive DAAT Query Processing

```
Open inverted lists for all query terms
Sort lists by length (shortest first)
for each doc_id in shortest list do
    Use seek() to find doc_id in all other lists
    if found in all lists then
        Score document
        Update top-k heap if score is high enough
    end if
end for
Return top-k results sorted by score
```

- This ensures $O(\log k)$ updates instead of sorting all results

Final results are sorted in descending order by score before returning.

4.4 Additional Components

Offset Index Builder: Creates a mapping from passage_id to byte offset in the collection file, enabling $O(1)$ random access to any passage without scanning the file. Required for snippet generation and displaying full passage text.

Snippet Generator: Extracts query-focused snippets from passages by finding all positions of query terms in passage text, selecting a window (default 200 chars) that includes the first query term, adjusting window boundaries to word boundaries, adding ellipsis if text is truncated, and highlighting query terms with HTML tags.

Web Interface: A Streamlit-based browser interface providing interactive search box, mode selection, adjustable number of results, snippet display with highlighted query terms, expandable full passage text, and index statistics display.

5 Performance and Index Statistics

5.1 Indexing Performance

Dataset: MS MARCO passage collection (8.8 million passages)

Metric	Value
Indexer Time	15-25 minutes
Indexer Memory	∼500 MB
Intermediate Blocks	50-100 files (∼200-400 MB)
Documents Processed	8,841,823
Unique Terms	∼2.5 million
Merger Time	10-20 minutes
Merger Memory	∼1-2 GB

Table 1: Indexing performance metrics

5.2 Final Index Size

Typical index sizes for full MS MARCO collection:

File	Size	Description
inverted_lists.bin	600-900 MB	Compressed posting lists
lexicon.bin	50-80 MB	∼2.5M terms + metadata
doc_table.bin	150-200 MB	8.8M document records
Total	**800-1200 MB**	4-6x compression

Table 2: Final index sizes

5.3 Query Performance

Query latency varies based on query term frequency, number of query terms, and query mode. Typical performance on a modern laptop:

Query Type	Terms	Latency
Disjunctive	1-2	10-50 ms
Disjunctive	3-4	50-200 ms
Disjunctive	5+	200-500 ms
Conjunctive	1-2	5-30 ms
Conjunctive	3-4	10-100 ms
Conjunctive	5+	50-300 ms

Table 3: Query performance by type and term count

Conjunctive queries are generally faster because they process fewer documents (only those containing all terms).

Memory usage during query processing:

- Fixed overhead: ∼300-500 MB (lexicon + document table)
- Per-query: ∼50-200 MB (inverted lists for query terms)

6 Design Decisions and Trade-offs

6.1 Language Choice

Python was chosen for rapid development, readable code, strong support for binary data manipulation, and easy library integration. While Python is slower than C++ or Java for CPU-intensive operations, with proper I/O optimization and efficient algorithms, performance is acceptable. The bottleneck is typically disk I/O, not CPU.

6.2 Compression Strategy

Varbyte encoding was selected over alternatives like Simple9 or PForDelta for its simplicity, fast encoding/decoding, good compression for small integers, and variable-length encoding that adapts to data distribution. Delta encoding for docIDs is essential for good compression (reduces integers by 2-3 orders of magnitude) with minimal encoding/decoding overhead.

6.3 Index Organization

Separate blocks for docIDs and frequencies enable better compression, selective decompression, and better cache locality during DAAT traversal. A single `inverted_lists.bin` file requires fewer file handles, simplifies file management, and enables better OS-level caching.

6.4 Memory Management

The full lexicon in memory enables $O(1)$ term lookup with reasonable memory usage (∼50-80 MB). The full document table in memory is required for BM25 scoring (need document lengths) and provides $O(1)$ passage_id lookup with reasonable memory usage (∼150-200 MB). Inverted lists are NOT loaded into memory as they are too large (∼600-900 MB total); they are loaded on-demand per query.

6.5 DAAT vs TAAT

Document-at-a-time (DAAT) was chosen over term-at-a-time (TAAT) for its memory efficiency (processes one document at a time), natural fit for top-k retrieval with heap, and early termina-

tion optimizations, despite requiring simultaneous traversal of multiple lists and more complex implementation.

7 Limitations and Future Improvements

7.1 Current Limitations

- **Passage Text Retrieval:** The command-line query processor only displays passage IDs by default; full passage text requires building the offset index
- **Scalability:** Full lexicon and document table must fit in memory; for collections > 100 million documents, would need multi-level lexicon or distributed architecture
- **No Caching:** Inverted lists are loaded from disk for every query
- **No Query Optimization:** Query terms processed in arbitrary order
- **Tokenization:** Simple alphanumeric tokenization with no handling of phrases, hyphenated words, stemming, or lemmatization
- **Ranking:** Only BM25 implemented; no learning-to-rank or neural ranking models

7.2 Possible Improvements

Performance:

- Implement static caching of most frequent terms
- Use impact-ordered indexes for faster top-k retrieval
- Implement MaxScore or WAND pruning for disjunctive queries
- Parallelize query processing across multiple cores

Compression:

- Try Simple9 or PForDelta for better compression ratios
- Use quantized impact scores instead of raw frequencies

Features:

- Add phrase query support (requires positional indexes)
- Implement query suggestions and spelling correction
- Add fielded search (title, body, metadata)

Quality:

- Implement learning-to-rank with neural models
- Add query expansion using word embeddings
- Support pseudo-relevance feedback

Scalability:

- Shard index across multiple machines
- Implement distributed query processing
- Use tiered indexes (memory + SSD + disk)

8 Testing and Validation

8.1 Correctness Testing

The system was validated for:

- **Tokenization:** Verified identical tokenization in indexer and query processor
- **Compression:** Verified encode/decode round-trip correctness with edge cases
- **Index Integrity:** Verified posting lists are sorted by doc_id and lexicon offsets point to correct inverted lists
- **Ranking:** Manually verified BM25 scores for sample queries
- **Query Processing:** Tested disjunctive and conjunctive modes with various queries

8.2 Test Queries

Sample queries used for testing:

1. "machine learning" - Tests multi-term query with common terms
2. "manhattan project" - Tests proper noun handling with medium frequency
3. "quantum computing artificial intelligence" - Tests longer query (4 terms)
4. "what is the capital of france" - Tests stopwords and natural language
5. Single-term queries: "python", "database", "algorithm"

8.3 Performance Testing

Measured query latency for various scenarios (different query lengths, modes, term frequencies, and result counts) to verify that query time scales reasonably with number of postings processed, conjunctive queries are faster than disjunctive for most queries, and top-k heap maintains correct ordering.

9 Conclusion

This project implements a complete search engine system capable of indexing and searching millions of text passages efficiently. The system demonstrates fundamental information retrieval concepts including I/O-efficient index construction using BSBI, compression techniques (varbyte and delta encoding), document-at-a-time query processing, BM25 ranking function, and both conjunctive and disjunctive query semantics.

The implementation prioritizes correctness, efficiency, and code clarity. Memory usage is carefully controlled through streaming algorithms and block-based processing. The compressed index achieves 4-6x compression while maintaining fast query times.

The modular design separates concerns cleanly: indexing, merging, and query processing are independent stages with well-defined inputs and outputs. This makes the system easy to understand, test, and extend.

Future work could focus on advanced optimizations (pruning, caching), additional ranking signals, and scalability to even larger collections. The current implementation provides a solid foundation for these enhancements.

A File Formats

A.1 Intermediate Block Format

Text format:

```
term [TAB] doc_id:freq,doc_id:freq,...

Example:
algorithm   0:2,15:1,23:3
database    5:1,12:2,18:1
```

A.2 Inverted List Binary Format

Header (12 bytes):

```
[doc_count: 4 bytes unsigned int]
[docid_length: 4 bytes unsigned int]
[freq_length: 4 bytes unsigned int]

Data:
[compressed delta-encoded docIDs: docid_length bytes]
[compressed frequencies: freq_length bytes]
```

A.3 Lexicon Binary Format

Header: `[num_terms: 4 bytes unsigned int]`
For each term:

```
[term_length: 4 bytes unsigned int]
[term: UTF-8 bytes]
[offset: 8 bytes unsigned long]
[length: 8 bytes unsigned long]
[doc_count: 4 bytes unsigned int]
```

A.4 Document Table Binary Format

Header: `[num_docs: 4 bytes unsigned int]`
For each document:

```
[doc_id: 4 bytes unsigned int]
[passage_id_length: 4 bytes unsigned int]
[passage_id: UTF-8 bytes]
[doc_length: 4 bytes unsigned int]
```