# Information Retrieval System: Indexer and Query Processor Documentation

## Abstract

This document describes the implementation of a scalable information retrieval system consisting of an indexer and query processor for Wikipedia entity data. The system implements memory-efficient indexing with external sorting, conjunctive document-at-a-time query processing with WAND optimization, and supports both TF-IDF and BM25 ranking functions. The implementation successfully processes 4.6M Wikipedia entities while operating within constrained memory limits through parallelization and disk-based merging strategies.

## 1. System Architecture and Data Structures

### 1.1 Indexer Architecture

The indexer implements a memory-constrained external sorting approach with the following core components:

**In-Memory Index Structure**: A nested dictionary `Dict[str, Dict[int, int]]` storing term → document → frequency mappings. This structure allows efficient accumulation of term frequencies during document processing while maintaining sorted order for efficient disk serialization.

**Document Index**: A dictionary `Dict[int, int]` mapping document IDs to document lengths (total term count). This structure supports BM25 scoring by providing document length normalization factors.

**Term Lexicon**: A dictionary `Dict[str, Dict]` containing metadata for each term including document frequency ( `df` ), file offset ( `offset` ), and entry length ( `length` ). This enables efficient random access to posting lists during query processing.

**Partial Index Management**: When memory usage exceeds 90% of the allocated limit (monitored via `psutil` ), the in-memory index is serialized to disk as a partial index file in JSONL format. Each partial contains sorted terms with their complete posting lists.

### 1.2 Query Processor Architecture

The query processor implements conjunctive document-at-a-time (DAAT) retrieval with WAND optimization:

**WAND Term Pointers**: Each query term is represented by a `WandTermPointer` containing sorted document IDs, term frequencies, and precomputed upper bounds for pruning. The pointer maintains an index for efficient traversal and skip-to operations.

**Scoring Functions**: Both TF-IDF and BM25 are implemented with standard formulations:

- TF-IDF: `(1 + log(tf)) × log(N/df)`
- BM25: `log((N-df+0.5)/(df+0.5)+1) × (tf×(k1+1))/(tf+k1×(1-b+b×dl/avgdl))`

## 2. Algorithms and Computational Complexity

### 2.1 Indexing Algorithm

The indexing process follows these steps:

1. **Document Processing**: Records are read in batches and processed in parallel using `multiprocessing.Pool` . Each worker parses, tokenizes, removes stopwords, and stems terms.
    - **Complexity**: O(D×L) where D is document count and L is average document length.
2. **Memory Management**: After processing each batch, memory usage is checked. When threshold is exceeded, the in-memory index is flushed to disk.
    - **Complexity**: O(T×log(T)) for sorting T unique terms per flush.
3. **External Merge**: All partial indexes are merged using a disk-based approach, reading one line at a time from each partial file and merging posting lists.
    - **Complexity**: O(P×T×log(P)) where P is the number of partials and T is total unique terms.

### 2.2 Query Processing Algorithm

Query processing implements the WAND algorithm for efficient top-k retrieval:

1. **Query Preprocessing**: Identical to document preprocessing (stopword removal, stemming).
    - **Complexity**: O(Q) where Q is query length.
2. **WAND Traversal**: Pointers are sorted by current document ID. Upper bound pruning determines pivot position. Documents are scored only when all terms align.
    - **Complexity**: O(K×T×log(T×P)) where K is result size, T is query terms, and P is average posting list length.

## 3. Implementation Details and Optimizations

### 3.1 Parallelization Strategy

**Indexing Parallelization**: Document parsing is parallelized across CPU cores using `multiprocessing.Pool`. Each worker independently processes document batches, with the main thread handling memory management and disk I/O. This design achieves near-linear speedup for CPU-bound parsing operations while maintaining thread safety for shared data structures.

**Memory Efficiency**: The implementation uses JSONL format for both partial indexes and final structures, enabling streaming I/O without loading entire indexes into memory. Posting lists are accessed on-demand during query processing through file seeking.

### 3.2 Text Processing Pipeline

The `RecordParser` class implements a standardized preprocessing pipeline using NLTK:

- **Tokenization**: Regex-based word boundary detection ( `\b\w+\b` )
- **Normalization**: Lowercase conversion
- **Stopword Removal**: English stopwords from NLTK corpus
- **Stemming**: Snowball stemmer for morphological normalization

This pipeline ensures consistency between indexing and query processing phases.

### 3.3 WAND Optimization

The WAND implementation uses NumPy arrays for efficient document ID and frequency storage, enabling vectorized operations. Skip-to operations use binary search within sorted arrays, reducing traversal complexity. Upper bound precomputation during index loading enables aggressive pruning during query processing.

## 4. Empirical Performance Analysis

The complete empirical results can be found on the file `performance_results.txt`.

### 4.1 Index Characterization

The complete Wikipedia entity corpus (4,641,784 documents) produces the following index statistics: Index Size: 1.9 GB (combined total from inverted_index.jsonl, document_index.json, and term_lexicon.json) Processing Time: 355 seconds (5 minutes 55 seconds) Unique Terms: 2,730,497 terms Average Posting List Length: 44.08 postings per term

This index size represents efficient compression through the use of JSONL, stopword removal, and stemming. Compared to the raw corpus size (2.1 GB), the index demonstrates competitive storage efficiency. The higher average posting list length (previously 3.91) now reflects improvements in corpus scale and document distribution.

### 4.2 Memory Management Efficiency

Using a 1GB memory limit, the indexer flushed 276 partial indexes to disk during processing. Memory usage remained under 1GB (peak observed 947MB) as monitored during processing. This confirms the success of memory-bound index partitioning using external sorting techniques and validates the system's scalability to large corpora.

### 4.3 Query Processing Performance

The system was tested using 5 benchmark queries, each executed using both TF-IDF and BM25 ranking functions:

#### TF-IDF Ranking Results

Average Score: 27.61 Score Range: 23.71 – 36.23 Average Results per Query: 10

#### BM25 Ranking Results

Average Score: 21.01 Score Range: 17.66 – 26.09 Average Results per Query: 10

#### Comparative Insights

TF-IDF generally produced higher scores compared to BM25. Consistent Top-10 Overlap between methods suggests stable retrieval across models. Result Coverage: 100% of queries returned results, demonstrating robust index coverage.

### 4.4 Ranking Function Insights (Selected Queries)

#### physics nobel winners since 2000

Top TF-IDF Score: 36.23 (Doc 2523905) Top BM25 Score: 25.93 (Doc 3056327) Shared Top-10 Doc IDs: 3056327, 2484928, 2475923, 2491835

#### christopher nolan movies

TF-IDF Top Score: 28.40 (Docs 0930716, 0930717) BM25 Top Score: 26.09 (Doc 2509879) Shared Top-10 Doc IDs: 0930716, 0930717, 2509879, 4131250, 1486638, 1975158

#### 19th century female authors

TF-IDF Top Score: 26.52 (Doc 2540654) BM25 Top Score: 19.07 (Doc 2646751) Shared Top-10 Doc IDs: 2646751, 2370608

#### german cs universities

TF-IDF Top Score: 23.71 (Doc 3523437) BM25 Top Score: 19.56 (Doc 0772638) Shared Top-10 Doc IDs: 0772638, 3523437, 0512918, 0772631

#### radiohead albums

TF-IDF Top Score: 30.23 (Doc 3442549) BM25 Top Score: 22.10 (Doc 2608716) Shared Top-10 Doc IDs: 2608716, 3442549, 4264708, 3442547, 3094193, 4095497, 3442552, 3442551, 1858807

### Summary

The revised statistics and retrieval outcomes reaffirm the robustness of the indexing engine and ranking models. TF-IDF tends to produce higher scores overall, while BM25 provides smoother term saturation handling. Both yield stable rankings and complete query coverage, confirming the system's reliability for large-scale information retrieval.

## 5. System Validation and Testing

### 5.1 Correctness Validation

The system includes comprehensive unit tests for the `Reader` class, validating thread safety, file handling, and edge cases. Integration testing confirms that indexes produced by `indexer.py` are correctly processed by `processor.py` with consistent results across multiple runs.

### 5.2 Scalability Testing

Testing with corpus subsets (10K, 100K, 1M documents) demonstrates linear scaling in both processing time and index size. Memory usage remains constant regardless of corpus size due to the external sorting approach.

## 6. Conclusion and Index Access

The implemented information retrieval system successfully addresses the core requirements of scalable indexing and efficient query processing. The combination of external sorting, parallel processing, and WAND optimization enables handling of large-scale document collections while maintaining reasonable response times and memory constraints.

The complete index structures generated from the Wikipedia entity corpus are available for download and evaluation at:
https://drive.google.com/file/d/1cV1dNTCXK_jWo9YHK_r5XayE7qGtsQNi/view?usp=sharing

Future enhancements could include additional compression schemes (gamma codes for document IDs), query parallelization, and advanced ranking functions incorporating entity-specific features. The modular design supports these extensions while maintaining the core efficiency characteristics demonstrated in this implementation.