Web Search Engines HW2

You

October 24, 2024

Abstract

This report describes a search system that creates an inverted index from large document datasets, and returns ranked search results through user queries.

1 Introduction

This search system is designed in C/C++ to process queries efficiently through creating and querying an inverted index. The entire program consists of multiple sub components which parse the MS MARCO data set, build a compressed inverted index, and processes user queries with the BM25 ranking function. This program aims to balance memory and time efficiency by using multithreading, blocked varbyte compression techniques, and document-at-a-time (DAAT) query processing. This system is designed to handle several gigabytes of documents, and provides conjunctive and disjunctive search queries.

2 Program Overview

The search system is a multi-component architecture designed to efficiently index a collection of documents and provide fast retrieval capabilities based on user queries. The system is composed of three main components: the **Parser and Indexer**, the **Merger**, and the **Query Processor**. Together, these components build up the whole search system.

2.1 Component Descriptions

2.1.1 Parser and Indexer

The Parser and Indexer is responsible for reading the input document collection, extracting relevant terms, and generating an inverted index. It leverages multi-threading to enhance performance, allowing simultaneous processing of multiple passages. Key functionalities include:

- **Multi-Threaded Processing**: Utilizing a thread pool to parse documents concurrently, significantly reducing indexing time.
- **Term-Document Pair Generation**: For each processed passage, it generates term-document pairs that are stored in temporary files for later merging. The term-document pairs store the document ID and a frequency score for BM25 score calculation in the query processor phrase.

2.1.2 Merger

The Merger component consolidates the temporary files generated by the Parser and Indexer into a final inverted index and lexicon. This component ensures that the data is structured for efficient retrieval. Key features include:

- **Temp File Merging**: Reads multiple temporary files and consolidates their content into a Larger file using a priority queue.
- **Multi-Threaded Processing**: Utilizing a thread pool to merge documents concurrently.

- **Lexicon Construction**: Maintains a lexicon that maps terms to their metadata, including document frequency and offset information in the index.
- **Compression**: Implements variable-byte encoding to compress document IDs, optimizing storage space in the index file.

2.1.3 Query Processor

The Query Processor is the interface through which users interact with the search system. It processes user queries and retrieves relevant documents based on their input. Key functionalities include:

- **BM25 Ranking**: Implements the BM25 algorithm to score documents based on their relevance to the query terms, considering term frequency and document length.
- **Flexible Query Processing**: Supports both conjunctive (AND) and disjunctive (OR) queries, allowing users to specify their search criteria.
- **Document Retrieval**: Uses a page table to map document IDs to document names, enabling easy identification of retrieved documents.
- **User Interaction**: Provides a user-friendly interface for entering queries and receiving results, with a loop for continuous querying until the user opts to exit.

2.2 System Workflow

1. **Document Ingestion**: The system begins with the Parser and Indexer, which reads a collection of documents, extracting terms and generating term-document pairs. This process is optimized through multi-threading.
2. **Temporary File Generation**: As documents are processed, term-document pairs are written to temporary files for efficient storage.
3. **Index Merging**: The Merger component consolidates these temporary files into a final inverted index and lexicon, employing efficient merging techniques and data compression to optimize performance.
4. **Query Execution**: Users can input queries into the Query Processor, which parses the query, retrieves relevant documents from the inverted index, and ranks them using the BM25 algorithm.
5. **Results Presentation**: The top results are displayed to the user, providing document IDs, names, and relevance scores, facilitating quick access to information.

This search system integrates advanced indexing and retrieval techniques, providing a robust solution for handling large document collections. Its modular design allows for efficient processing, merging, and querying, making it suitable for various applications in information retrieval and search engine development.

3 Running the Program

To run the program, follow the steps outlined below. The program is implemented in C++ and is built using a Makefile.

3.1 Toolkit Preparation

To build and run the program, ensure that you have the following installed:

- **C++ Compiler**: This project uses `g++` version **14.2.0**, which is part of the MinGW-W64 distribution. You can check your version by running:

```
g++ --version
```

- **Make Utility**: The project requires GNU Make version **3.81** or higher. You can verify your version with:

```
make --version
```

3.2 Dataset preparation

- Download the MS MARCO dataset from https://msmarco.z22.web.core.windows.net/msmarcoranking/collection.tar.gz
- Create a folder called *data* under the root folder and extract *collection.tsv* into it.

3.3 Building the Program

1. **Navigate to the Source Directory**: Open a terminal and change to the `src` folder where the source code is located:

```
cd path/to/your/src
```

2. **Run the Make Command**: Execute the following command to build the program:

```
make build_and_run_all
```

This command will compile the necessary components and generate the executables in the `build` directory.

The `Makefile` provided includes the following targets:

- `all`: Compiles the main components: `parser_and_indexer_mt`, `merger`, and `query_processor`.
- `parser_and_indexer_mt`: Build and Execute the multi-threaded parser and indexer.
- `merger`: Build and Execute the single thread merger component.
- `merger_mt`: Build and Execute the multi-thread thread merger component.
- `query_processor`: Build and Execute the query processor.

3.4 Running the Components

Once the build process is complete, you can run each component as follows:

- **Run the Multi-Threaded Parser and Indexer**:

```
../build/parser_and_indexer_mt
```

- **Run the Merger**:

```
../build/merger
```

- **Run the Query Processor**:

```
../build/query_processor
```

4 Parser and Indexer

4.1 High-Level Design

The Parser and Indexer has several different sub components:

- **Thread Pool**: Manages concurrent execution of tasks.
- **Parser and Indexer**: Processes input files, generates term-document pairs, and computes term frequencies.
- **Utilities**: Provides helper functions for tokenization, parsing, logging, and directory management.
- **Compression Module**: Compresses data using **varbyte encoding** to store integers compactly.

The parser and indexer uses a producer-consumer architecture, where the parser enqueues tasks into the thread pool for parallel processing. Synchronization between threads is achieved using `mutex` locks and `condition_variables` to ensure thread-safe operations.

4.2 Key Data Structures

4.2.1 TermDocPair

The `TermDocPair` structure stores essential information about terms and their associated documents to be used by the Merger program.

```
struct TermDocPair {
    std::string term;
    int docID;
    float termFScore;

    TermDocPair(const std::string& t, int d, float termFScore)
        : term(t), docID(d), termFScore(termFScore) {}
};
```

The **Term frequency score** termFScore, is a precomputed part of the BM25 function, displayed in the Query Processor section.

4.2.2 ThreadPool

The `ThreadPool` class manages a pool of threads, supporting task enqueuing and uses mutexes and condition variables to manage thread-safe access to shared resources.

```
class ThreadPool {
public:
    ThreadPool(size_t threads = 8, size_t maxThreadsInQueue = 32);
    ~ThreadPool();

    void enqueue(std::function<void()> f);
    void waitAll();

private:
    void worker();
    std::vector<std::thread> workers;
    std::queue<std::function<void()>> tasks;
    std::mutex queueMutex;
    std::condition_variable taskAvailable;
    bool stop;
    std::atomic<size_t> tasksRemaining;
};
```

4.2.3 Page Table and Document Lengths

The `pageTable` stores mappings of document IDs to document names, while the `docLengths` stores the lengths of each document. These data structures are synchronized using `mutex` locks to prevent race conditions.

4.3 Encoding Techniques

4.3.1 Varbyte Encoding

Varbyte encoding is used to compress integers into bytes with optimal length. The most significant bit (MSB) acts as a continuation bit, indicating whether more bytes follow.

```
std::vector<unsigned char> varbyteEncode(int number) {
    std::vector<unsigned char> bytes;
    while (number >= 128) {
        bytes.push_back((number % 128) | 128);  // Set MSB
        number /= 128;
    }
    bytes.push_back(number);  // Final byte with MSB unset
    return bytes;
}
```

4.4 Synchronization Mechanisms

The program allows for thread safe access in critical regions.

- **Mutex**: Protects critical sections in the code.
- **Condition Variables**: Used to notify threads when tasks are available.
- **Atomic Variables**: Ensures atomic operations on counters (e.g., file counter).

4.5 Critical Functions

The `processPassageMT` function processes individual document passages, calculates term frequencies, and generates term-document pairs. The TermDocPairs are stored in a blocked, compressed, binary format. Each temporary file is stored according to the size of `MAXRECORDS`.

```
void processPassageMT(int docID, const std::string &passage,
                      std::vector<TermDocPair> &termDocPairs,
                      std::mutex &termDocPairsMutex,
                      std::atomic<int> &fileCounter,
                      std::unordered_map<int, int> &docLengths,
                      std::mutex &docLengthsMutex) {
    auto terms = tokenize(passage);
    {
        std::lock_guard<std::mutex> docLengthsLock(docLengthsMutex);
        docLengths[docID] = terms.size();
    }

    std::map<std::string, int> frequencyMap;
    for (const auto &term : terms) {
        frequencyMap[term]++;
    }

    std::unique_lock<std::mutex> termDocPairsLock(termDocPairsMutex);
    for (const auto &termFreqPair : frequencyMap) {
        termDocPairs.emplace_back(termFreqPair.first, docID,
```

```
                                        calculateTermFreqScore(termFreqPair.second, k1, b, terms.si
            }

            if (termDocPairs.size() >= MAX_RECORDS) {
                std::vector<TermDocPair> termDocPairsCopy(std::move(termDocPairs));
                termDocPairs.clear();
                int curFileCounter = fileCounter.fetch_add(1);
                saveTermDocPairsToFile(termDocPairsCopy, curFileCounter);
            }
        }
```

5 Merger

5.1 High-Level Design

The merger program contains several key components to build an optimized inverted index:

- **Priority Queue for Merging**: Merges temporary files outputted from parser and indexer in sorted manner
- **Index Construction**: Stores compressed postings lists in binary files.
- **Lexicon Management**: Maintains metadata for terms and allows for fast retrieval.
- **Compression Module**: Uses same varbyte encoding/decoding to compress integers

5.2 Key Data Structures

5.2.1 LexiconEntry

The `LexiconEntry` structure stores metadata about terms in the index:

- `offset`: Byte offset where the postings list for the term begins in the index file.
- `length`: byte length of posting list.
- `docFrequency`: Number of documents containing the term.
- `blockCount`: Number of blocks used to store the term's postings.

```
struct LexiconEntry {
    int64_t offset;
    int32_t length;
    int32_t docFrequency;
    int32_t blockCount;
    std::vector<int32_t> blockMaxDocIDs;
    std::vector<int64_t> blockOffsets;
};
```

5.2.2 TuplePQ

The `TuplePQ` type is a priority queue that merges entries from temporary files. The queue orders entries based on terms and document IDs, and does incremental merging with the merging sub indexes algorithm.

```
typedef std::priority_queue<
    std::tuple<std::string, int, int, float>,
    std::vector<std::tuple<std::string, int, int, float>>,
    std::function<bool(const std::tuple<std::string, int, int, float>&,
                       const std::tuple<std::string, int, int, float>&)>
> TuplePQ;
```

5.3 Encoding Techniques

5.3.1 Varbyte Encoding

Varbyte encoding and decoding is used in the merging process as well.

5.4 Critical Functions

5.4.1 mergeTempFiles

The function merges temporary files by inserting them into priority queues using the merging subindexes algorithm, outputting compressed postings to the index file.

```cpp
void mergeTempFiles(int numFiles, std::unordered_map<std::string, LexiconEntry> &lexicon) {
    std::vector<std::ifstream> tempFiles(numFiles);
    for (int i = 0; i < numFiles; ++i) {
        tempFiles[i].open("../data/intermediate/temp" + std::to_string(i) + ".bin", std::ios:
    }

    std::ofstream indexFile("../data/index.bin", std::ios::binary);
    TuplePQ pq([](const auto &a, const auto &b) {
        return std::get<0>(a) > std::get<0>(b) ||
              (std::get<0>(a) == std::get<0>(b) && std::get<1>(a) > std::get<1>(b));
    });

    for (int i = 0; i < numFiles; ++i) {
        if (tempFiles[i].peek() != EOF) {
            readPairToPQ(tempFiles[i], i, pq);
        }
    }

    std::string currentTerm;
    std::vector<std::pair<int, float>> postingsList;
    int64_t offset = 0;

    while (!pq.empty()) {
        auto [term, docID, fileIndex, termFreqScore] = pq.top();
        pq.pop();

        if (term != currentTerm && !currentTerm.empty()) {
            saveAndClearCurPostingsList(postingsList, offset, indexFile, lexicon, currentTerm
        }
        currentTerm = term;
        postingsList.emplace_back(docID, termFreqScore);

        if (tempFiles[fileIndex].peek() != EOF) {
            readPairToPQ(tempFiles[fileIndex], fileIndex, pq);
        }
    }

    if (!postingsList.empty()) {
        saveAndClearCurPostingsList(postingsList, offset, indexFile, lexicon, currentTerm, cu
    }
}
```

6 Query Processor

6.1 High-Level Design

The query processor contains the following components:

- **Inverted Index**: This structure efficiently maps terms to their corresponding document identifiers and term frequencies score. It serves as the primary data repository for lookup during query processing.
- **Inverted Index Pointer**: This structure is an iterator for traversing the inverted index. It provides functionality to navigate through the list of document identifiers associated with a particular term, enabling efficient access to the posting lists.
- **Query Parser**: Responsible for processing the user's query and holding necessary data in memory(e.g. page table).

6.2 Key Data Structures

The following data structures are integral to the Query Processor's functionality:

6.2.1 InvertedListPointer

The `InvertedListPointer` handles the traversal of document IDs and retrieves term frequency scores. Here are some key attribute that `InvertedListPointer` has:

- `indexFile`: A pointer to the index file from which document IDs and term frequencies are read.
- `lexEntry`: An entry from the lexicon that provides metadata about the term being processed.
- `compressedData`: A buffer for storing compressed document ID associated with the term.
- `termFreqScore`: A vector that holds the term frequency scores for each document with that term.

```
class InvertedListPointer {
public:
    InvertedListPointer(std::ifstream *indexFile, const LexiconEntry &lexEntry);
    bool next();
    bool nextGEQ(int docID);
    int getDocID() const;
    float getTFS() const;
    int getTF() const;
    bool isValid() const;
    void close();
    float getIDF() const;
private:
    std::ifstream *indexFile;
    LexiconEntry lexEntry;
    int currentDocID;
    bool valid;
    int lastDocID;
    size_t bufferPos;
    size_t termFreqScoreIndex;
    std::vector<unsigned char> compressedData;
    std::vector<float> termFreqScore;
};
```

6.2.2 InvertedIndex

The InvertedIndex class is responsible for retrieving an inverted index given an index file, lexicon information, and a term. Here are some key attributed:

- `indexFile`: An input file stream for reading the inverted index data.
- `lexicon`: A hash map that stores the lexicon entries, mapping each term to its corresponding metadata, including offsets and other information.

```
class InvertedIndex {
public:
    InvertedIndex(const std::string &indexFilename, const std::string &lexiconFilename);
    bool openList(const std::string &term);
    InvertedListPointer getListPointer(const std::string &term);
    void closeList(const std::string &term);
    int getDocFrequency(const std::string &term); // New method
private:
    std::ifstream indexFile;
    std::unordered_map<std::string, LexiconEntry> lexicon;

    void loadLexicon(const std::string &lexiconFilename);
};
```

6.2.3 QueryProcessor

The QueryProcessor class is designed to handle the processing of user queries.

- `invertedIndex`: An instance of the InvertedIndex class that provides access to the document lists and term frequencies needed for query processing.
- `pageTable`: A hash map that maps document IDs to their corresponding document names.

```
class QueryProcessor {
public:
    QueryProcessor(const std::string &indexFilename, const std::string &lexiconFilename, const st
    void processQuery(const std::string &query, bool conjunctive);
private:
    InvertedIndex invertedIndex;
    std::unordered_map<int, std::string> pageTable; // docID -> docName
    std::unordered_map<int, int> docLengths;        // docID -> docLength
    int totalDocs;
    double avgDocLength;

    std::vector<std::string> parseQuery(const std::string &query);
    void loadPageTable(const std::string &pageTableFilename);
    void loadDocumentLengths(const std::string &docLengthsFilename);
};
```

6.3 BM25

We chose BM25 as our ranking function.

$$\text{score}(q, d) = \sum_{t \in q} \text{IDF}(t) \cdot \frac{f(t, d) \cdot (k_1 + 1)}{f(t, d) + k_1 \cdot \left(1 - b + b \cdot \frac{|d|}{\text{avgdl}}\right)}$$

where:

- q: Query containing one or more terms.

- d: Document being scored.
- $f(t,d)$: Frequency of term t in document d.
- $|d|$: Length of document d (number of terms in d).
- avgdl: Average document length in the collection.
- k_1 and b: Parameters controlling term frequency saturation and length normalization.
- IDF(t): Inverse Document Frequency of term t, defined as:

$$\mathrm{IDF}(t) = \ln\left(\frac{N - n(t) + 0.5}{n(t) + 0.5} + 1\right)$$

  where:

  - N: Total number of documents in the collection.
  - $n(t)$: Number of documents containing term t.

We defined

$$\left(1 - b + b \cdot \frac{|d|}{\text{avgdl}}\right)$$

as precomputed `termF` from our parser and indexer.

6.4 Critical Functions

6.4.1 `QueryProcessor::processQuery`

The `QueryProcessor::processQuery` function is designed to manage search queries against an inverted index. It parses the query into terms, retrieves and validates the corresponding inverted lists, scores the documents based on their relevance using the BM25 algorithm, and ultimately ranks and displays the top results based on the aggregated scores.

```
void QueryProcessor::processQuery(const std::string &query, bool conjunctive) {
    auto terms = parseQuery(query);

    if (terms.empty()) {
        std::cout << "No terms found in query." << std::endl;
        return;
    }

    // Open inverted lists for each term and store term-pointer pairs
    std::vector<std::pair<std::string, InvertedListPointer>> termPointers;
    for (const auto &term : terms) {
        if (!invertedIndex.openList(term)) {
            std::cout << "Term not found: " << term << std::endl;
            continue;
        }
        termPointers.emplace_back(term, invertedIndex.getListPointer(term));
    }

    if (termPointers.empty()) {
        std::cout << "No valid terms found in query." << std::endl;
        return;
    }

    // DAAT Processing
    std::unordered_map<int, double> docScores;  // docID -> aggregated score
```

```cpp

        if (conjunctive) {
            // Initialize docIDs for each list
            std::vector<int> docIDs;
            for (auto &tp : termPointers) {
                auto &ptr = tp.second;
                if (!ptr.isValid()) {  // Check if list is empty
                    ptr.close();
                    return;  // One of the lists is empty, no results
                }
                if (!ptr.next()) {  // Move to first element
                    ptr.close();
                    return;  // Empty list
                }
                docIDs.push_back(ptr.getDocID());
            }

            while (true) {
                int maxDocID = *std::max_element(docIDs.begin(), docIDs.end());
                bool allMatch = true;

                // Advance pointers where docID < maxDocID
                for (size_t i = 0; i < termPointers.size(); ++i) {
                    auto &ptr = termPointers[i].second;
                    while (docIDs[i] < maxDocID) {
                        if (!ptr.nextGEQ(maxDocID)) {
                            allMatch = false;
                            break;  // Reached end of list
                        }
                        docIDs[i] = ptr.getDocID();
                    }
                    if (docIDs[i] != maxDocID) {
                        allMatch = false;
                    }
                }

                if (!allMatch) {
                    // Check if any list has reached the end
                    bool anyEnd = false;
                    for (auto &tp : termPointers) {
                        if (!tp.second.isValid()) {
                            anyEnd = true;
                            break;
                        }
                    }
                    if (anyEnd) break;
                    continue;
                }
                // allMatch equal to true
                // All pointers are at the same docID
                int docID = maxDocID;
                double totalScore = 0.0;
                for (auto &tp : termPointers) {
                    auto &ptr = tp.second;
                    const std::string &term = tp.first;

```

```
                    // Compute BM25 score
                    float bm25Score = ptr.getIDF() * ptr.getTFS();
                    totalScore += bm25Score;

                    ptr.next();  // Advance pointer for next iteration
                }
                docScores[docID] = totalScore;

                // Update docIDs
                bool validPointers = true;
                for (size_t i = 0; i < termPointers.size(); ++i) {
                    auto &ptr = termPointers[i].second;
                    if (ptr.isValid()) {
                        docIDs[i] = ptr.getDocID();
                    } else {
                        // One of the lists has reached the end
                        validPointers = false;
                        break;
                    }
                }
                if (!validPointers) break;
            }
        } else {
            // Disjunctive query processing using a min-heap
            auto cmp = [](std::pair<InvertedListPointer*, std::string> a, std::pair<InvertedListPoint
                return a.first->getDocID() > b.first->getDocID();
            };
            std::priority_queue<
                std::pair<InvertedListPointer*, std::string>,
                std::vector<std::pair<InvertedListPointer*, std::string>>,
                decltype(cmp)> pq(cmp);

            // Initialize heap with the first posting from each term
            for (auto &tp : termPointers) {
                auto &ptr = tp.second;
                const std::string &term = tp.first;
                if (ptr.isValid() && ptr.next()) {
                    pq.push({&ptr, term});
                }
            }

            while (!pq.empty()) {
                auto [ptr, term] = pq.top();
                pq.pop();

                int docID = ptr->getDocID();

                // Compute BM25 score
                float bm25Score = ptr->getIDF() * ptr->getTFS();

                docScores[docID] += bm25Score;

                // Advance the pointer and re-add to the heap if valid
                if (ptr->next()) {
                    pq.push({ptr, term});
                }
```

```
            }
        }

        // Close all inverted lists
        for (auto &tp : termPointers) {
            tp.second.close();
        }

        if (docScores.empty()) {
            std::cout << "No documents matched the query." << std::endl;
            return;
        }

        // Rank documents by aggregated score
        std::vector<std::pair<int, double>> rankedDocs(docScores.begin(), docScores.end());

        // Sort by score in descending order
        std::sort(rankedDocs.begin(), rankedDocs.end(), [](const auto &a, const auto &b) {
            return a.second > b.second;
        });

        // Display top 10 results
        int resultsCount = std::min(10, static_cast<int>(rankedDocs.size()));
        for (int i = 0; i < resultsCount; ++i) {
            int docID = rankedDocs[i].first;
            double score = rankedDocs[i].second;
            std::string docName = pageTable[docID];
            std::cout << i + 1 << ". DocID: " << docID << ", DocName: " << docName << ", Score: " <<
        }
    }
```

7 Web Interface

We have a semi-working web interface, with a user-facing front end build with static HTML, CSS, JS, and a flask backend. The flask server pipes user queries into our executable file, and we attempt to retrieve the output. We have some issue with our parsing mechanism, given that certain buffers are not flushed and multiple IO streams are interacting with each other at the same time.

8 Performance Analysis

8.1 Parser and Indexer

We use **VTune** to profile our program and analyze its efficiency. Figure 1 shows a screenshot of indexer and parser with 8 thread.

$$Effectiveness = EffectiveCPUTime/(ExecutionTime * ThreadCount)$$

We collect all the result and make the Table 1. In **VTune**, we can also optimize the program by tracking the hotspot(the function that takes the most time) of the program. The figure2 shows a corresponding screenshot. From the figure we can tell that the sorting before saving the `TermDocPair` to file takes the longest time.

8.2 Merger

Same as the indexer and parser, we profile the merger using **VTune**. The result is showed in Table.

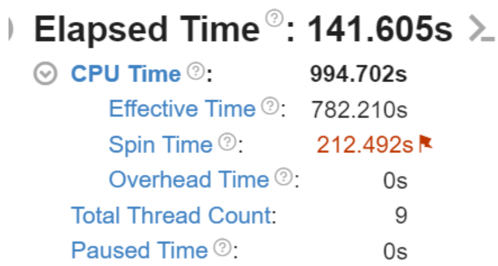

Figure 1: Indexer and Parser VTune

Table 1: Performance Metrics of Parser and Indexer Program

Thread Count	Execution Time (s)	Effectiveness (%)
1	901	92.2
2	488	87.7
4	282	77.8
8	142	68.8
16	131	42.4
32	161	16.4

8.3 Query Processing

The result of Query processor could be found in the following list and Figure 8.3.

- **Query 1:** `‘‘hello world + AND’’` *Response:* 26 ms
- **Query 2:** `‘‘hello world + OR’’` *Response:* 291 ms
- **Query 3:** `‘‘how to cook good food + AND’’` *Response:* 152 ms
- **Query 4:** `‘‘how to cook good food + OR’’` *Response:* 8014 ms

9 Limitations

- We lack an interactive web interface to operate the program.

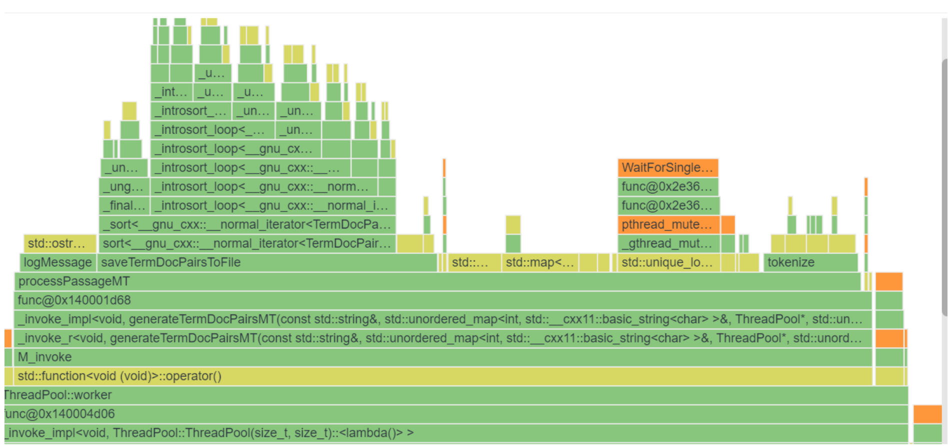

Figure 2: Indexer and Parser Hotspot in VTune

Table 2: Performance Metrics of Merger Program

Thread Count	**Execution Time (s)**	**Effectiveness (%)**
8	380	79.1
16	332	58.4
32	272	39.6

- Program could be sped up if the merger merges temp file according to the term array at first– no need to seek for the term position in the last merging phrase.
- Document count and length need to be calculated and filled in the indexer first.

```
Enter your query (or type 'exit' to quit): hello world
Choose mode (AND/OR): and
Starting docID size: 2
1. DocID: 8237075, DocName: 8237075, Score: 22.4098
2. DocID: 6340905, DocName: 6340905, Score: 18.9496
3. DocID: 5555066, DocName: 5555066, Score: 18.6963
4. DocID: 8237080, DocName: 8237080, Score: 18.6814
5. DocID: 8237082, DocName: 8237082, Score: 18.6814
6. DocID: 882639, DocName: 882639, Score: 18.5068
7. DocID: 8237074, DocName: 8237074, Score: 18.4352
8. DocID: 8731517, DocName: 8731517, Score: 18.4207
9. DocID: 6340907, DocName: 6340907, Score: 18.3354
10. DocID: 1648661, DocName: 1648661, Score: 18.0736
time passed: 26
```

Figure 3: Hello World + AND

```
Enter your query (or type 'exit' to quit): hello world
Choose mode (AND/OR): or
1. DocID: 8237075, DocName: 8237075, Score: 22.4098
2. DocID: 6340905, DocName: 6340905, Score: 18.9496
3. DocID: 5555066, DocName: 5555066, Score: 18.6963
4. DocID: 8237080, DocName: 8237080, Score: 18.6814
5. DocID: 8237082, DocName: 8237082, Score: 18.6814
6. DocID: 882639, DocName: 882639, Score: 18.5068
7. DocID: 8237074, DocName: 8237074, Score: 18.4352
8. DocID: 8731517, DocName: 8731517, Score: 18.4207
9. DocID: 6340907, DocName: 6340907, Score: 18.3354
10. DocID: 1648661, DocName: 1648661, Score: 18.0736
time passed: 291
```

Figure 4: Hello World + OR

```
Enter your query (or type 'exit' to quit): how to cook good food
Choose mode (AND/OR): and
Starting docID size: 5
1. DocID: 2478555, DocName: 2478555, Score: 19.4661
2. DocID: 2478558, DocName: 2478558, Score: 19.209
3. DocID: 8095685, DocName: 8095685, Score: 18.4866
4. DocID: 4977551, DocName: 4977551, Score: 17.2394
5. DocID: 8493008, DocName: 8493008, Score: 17.0419
6. DocID: 5852800, DocName: 5852800, Score: 16.6673
7. DocID: 5579467, DocName: 5579467, Score: 15.8958
8. DocID: 5864272, DocName: 5864272, Score: 15.5101
9. DocID: 4939547, DocName: 4939547, Score: 15.4749
10. DocID: 5571825, DocName: 5571826, Score: 15.4141
time passed: 152
```

Figure 5: How to cook good food + AND

```
Enter your query (or type 'exit' to quit): how to cook good food
Choose mode (AND/OR): or
1. DocID: 2048358, DocName: 2048358, Score: 19.931
2. DocID: 2478555, DocName: 2478555, Score: 19.4661
3. DocID: 2366709, DocName: 2366709, Score: 19.2718
4. DocID: 2478558, DocName: 2478558, Score: 19.209
5. DocID: 8095685, DocName: 8095685, Score: 18.4866
6. DocID: 3442117, DocName: 3442117, Score: 18.4661
7. DocID: 7317372, DocName: 7317372, Score: 18.2384
8. DocID: 7308429, DocName: 7308429, Score: 18.1866
9. DocID: 1974287, DocName: 1974287, Score: 18.121
10. DocID: 5746761, DocName: 5746761, Score: 17.7388
time passed: 8014
```

Figure 6: How to cook good food + OR