Department of Information Engineering
Artificial Intelligence and Data Engineering
Multimedia Information Retrieval and Computer Vision

MS-MARCO Search Engine

E. Ruffoli, F. Hudema

Academic Year 2022/2023

Contents

1 Application Overview

In this report, we present a simple search engine that aims to provide fast and accurate search results for the MS Marco document collection, a large dataset of web pages that is commonly used for research in information retrieval.

The application is composed by an indexer that creates an inverted index, which is a data structure that allows efficient search by mapping terms to the documents that contain them. The application also has a query executor that processes the user queries and returns the search results; to rank the search results the BM25 scoring function is used. Both conjunctive and disjunctive queries are supported. The search engine also employes dynamic pruning techniques to speed up query processing. In addition to providing fast search results, the search engine also aims to be performant in terms of memory usage. To achieve this, it uses simple compression techniques to compress the inverted index while maintaining fast search times. To make the application easy to use, a simple command line interface is provided to the users to execute queries or to create the inverted index.

In the following sections, the design and implementation of each of these components will be presented in detail and the performance of the search engine will be evaluated using standard metrics from the information retrieval literature. In the final section will be analyzed the limitations of the search engine.

2 Application Modules

The application can be divided in 5 different modules that will be analyzed in detail in the following paragraphs.

2.1 Parsing Phase

The parsing phase is the first phase of the indexing process. During the parsing, the document collection is divided into blocks, so that a single block can be stored in main memory, and a partial inverted index is built for each of these block. At the end of this phase a list of intermediate files, each of which contains a partial inverted index, and the document table data structure are created.

2.1.1 Compressed Reading

At the begininning of the parsing phase, the gzip compressed file containing the collection of documents is read and decompressed "on the fly", so that we do not have to decompress the whole collection at once, reducing the used disk space. The compressed reading is done using *boost::streams::filtering_streambuf* class that uses Zlib library to decompress on the fly gzip files. The compressed gzip file is also memory mapped, using *boost::iostreams::mapped_file_source mmap()*, to speed up the reading process.

2.1.2 SPIMI

The document collection is processed in a Block Sorted Based Indexing (BSBI) fashion using Single Pass Memory Indexing (SPIMI) to process each block; we do not employ regular BSBI_invert so as not to have to store the *term-termid mapping*.

Note that, since the function that tracks memory usage provided by the windows.h library is volatile and not stable due to the dynamic nature of memory allocation of the Windows operative system, interference with system processes and the presence of background processes, we can not use it in the SPIMI process. As a solution, we add an input parameter to set the number of documents in each block; however the SPIMI version that checks the memory usage is still present in commentated form in the *parsing.cpp* file (lines 4-21 and 267).

In order to optimize the indexing time, each block of documents is processed using a *thread pool*, a collection of worker threads. The main advantage of using a thread pool is that it can help to reduce the overhead associated with creating and destroying threads. When a task needs to be executed, it is added to a queue, and a worker thread from the pool is responsible for executing the task. We used the following thread pool library for C++ 17 `https://github.com/bshoshany/thread-pool`; this library also has a very useful *parallelize loop* function that partitions a loop in an orderly fashion and assign each part to a worker thread.

After each thread has finished processing it returns the calculation results on the portion of documents that has been assigned to it by two partial data structures:

- *dict*, that contains the term and its posting list in the form of a list of tuples (doc_id, freq);
- *partial doc table*, that contains the number of words of each document of the portion.

Once all threads have finished the processing, the partial outputs of each thread are merged in the final output of the current block. The merging of the *dict* is quite efficient because since the documents are assigned to each thread in sorted order by *doc_id*, to build the final partial index of the block we just need to concatenate the output of the thread that processed the first portion of the block, then the thread that processed the second portion and so on. Since we used the std::list data structure this concatenation operation is done in O(1) time complexity using the splice() function. The *partial doc tables*, that once again are sorted by *doc_id*, are written in the final document table file using the *insert* function.

At the end of the processing of a block, the final partial inverted index, lexicographically ordered, is written on an intermediate file with the following form:

<term posting_list>

2.1.3 Tokenization

Regarding the tokenization of the documents, that is carried out by the worker threads, it is done in the following steps:

1. Replace characters that are not numbers or ASCII letters with spaces; Note that punctuation and tabs are replaced with spaces;
2. Transform each upper case letter to the corresponding lower case one;
3. Split the document text using as separator character the whitespace; each non empty component of the splitted text is a token;
4. Discard the token if it is a stopword;
5. Stem the token using Porter2 algorithm;

2.2 Merging Phase

The merging phase is the second phase of the indexing process, it takes as input the intermediate files produced by the parsing phase, and outputs the final compressed inverted index, and the lexicon data structure.

2.2.1 Partial Index Merging

The merging of the partial inverted indexes stored in the intermediate files is done following a multi-way merge: all intermediate files are opened and a read buffer is maintained for each of them; also a write buffer is maintened to write on file the final inverted index. By using a priority_queue, built to behave like a min_heap, we select at each iteration the lowest term in lexicographical order that has not been processed yet; then all the posting lists belonging to that term are merged and the final posting is written on the inverted index file in compressed form. The offset at which the current posting list has been written are stored in the corresponding term entry of the Lexicon data structure.

2.2.2 Index Compression

As previously mentioned, after having joined each of the intermediate posting lists of the current term, the resulting posting list is written on file in compressed form. The doc_ids and the frequencies are stored in different files.

We chose Variable byte (VB) encoding as the method for compressing integers as it provides very good decompression performances. We encode the doc_ids as *dgaps* while the frequencies are encoded as they are.

2.2.3 Dynamic Pruning

In order to speed up the posting lists processing, the search engine adopts dynamic pruning techniques: in particular the MaxScore algorithm.

As far as the skip pointers concerns, a list of skip pointers is built only if a term has a number of postings higher than 1024. The skip pointers are stored in the doc_ids inverted index file, at the beginning of each posting list with length higher than 1024. A single skip pointer has the following structure:

<max_doc_id, doc_id_offset, freqs_offset>

max_doc_id is the maximum doc_id in the block pointed by the skip pointer (and so the last element of the block since they are sorted by doc_id), *doc_id_offset* points to the start of the block on the doc_id files and *freqs_offset* points to the start of the block on the frequencies file.

Note that, in order to encode smaller integers and save more space, the starting offsets of each block are "relative" offsets that need to be added to the corresponding posting list offset retrieved from the Lexicon. Since in the doc_ids file there are also the skip pointers at the beginning of each posting list, to retrieve the actual start of the block on the doc_id files we also need to add the byte size of the skip pointers list.

MaxScore also requires that some precomputations are performed: whenever a final posting list of a term is generated, we also need to compute the term upper bound value. The precomputed value will be stored in the Lexicon at the corresponding term entry and will be retrieved during the posting list processing to efficiently compute the MaxScore algorithm.

2.3 Query Execution

Query execution is the process of running a query against a collection of documents to retrieve the desired information and it is performed by the query processor module. The results of the query are presented to the user as a ranked list of documents sorted by relevance score.

2.3.1 Posting List Iterator

The query processor exploits the *Posting List Iterator*, an object that provides a simple interface with the following operations:

- *openList(docs_offset, freqs_offset, posting_list_len)*: opens the *inverted index docs* and *inverted index freqs* files, positions the file pointers at the corresponding offsets passed by parameters. It also reads and store the skip pointers if present;
- *closeList()*: closes the *inverted index docs* and *inverted index freqs* files;
- *next()*: moves the iterators to the next posting, decompress the doc_id and the frequency and store them in the *cur_doc_id* and *cur_freq* member variables. If the iterator has reached the end of the posting list return the end of list marker;
- *nextGEQ(d)*: advances the iterators forward to the next posting with a doc_id greater than or equal to *d*. If the current posting's doc_id is greater or equal to *d*, the *cur_doc_id* and *cur_freq* are left unchanged. If the current posting's doc_id is lower than *d*, sets the file pointers to the start of the block that has a max_doc_id greater or equal than *d*, and moves the iterators using the *next()* until we reach a doc_id that is greater than or equal to *d*. If *d* is greater than the doc_id of the last posting returns the end of list marker;
- *getDocid()*: returns the current doc_id of the posting list;
- *getFreq()*: returns the current freq of the posting list;

In this way all the compression and query pruning logic below is hidden from the higher-level of query processor and are handled inside the *Posting List Iterator*.

2.3.2 Scoring Function

This program implements BM25 (Best Matching 25), but also have an implementation of TF-IDF (Term Frequency-Inverse Document Frequency) that can be used. They are two common functions that are used in information retrieval to rank the relevance of documents in response to a query.

The BM25 parameters are set to k1=0.82, b=0.68; these values were taken from the following repository that performed grid search tuning on the MS Marco collection.

2.3.3 Types of Query

In this system queries can be processed in *conjunctive* (AND) or in *disjuntive* (OR) mode. A conjunctive query must return a list of documents in which all the terms of the query appears at least once, instead a disjunctive query must return all documents in which at least one query term appears.

The query execution can be performed in three different execution modes:

- *conjunctive_mode*: it is implemented using the Culpepper and Moffat's max algorithm;
- *disjunctive_mode*: it is implemented using the DAAT algorithm;
- *disjunctive_mode_MaxScore*: it is implemented using DAAT with the MaxScore optimisation;

2.4 Data Structures

A search engine typically has to work with a large amount of data, and storing this data in an external file can allow the search engine to access and manipulate the data without having to load it all into memory at once. For this reason the Lexicon and the Document Table are implemented as external file data structures and memory mapped using the Boost library.

2.4.1 Document Table

The Document Table implementation follows the one of a external file based vector. It also contains some informations that will be useful to compute the BM25 scoring function such as Document Table length and average document length. An entry of the Document Table has the following format:

<doc_no, doc_len>

The doc_no element is a string that is truncated on 10 bytes in order to have a fixed size entry. The Document Table has the following member functions:

- *create(filename)*: creates a document table at the specified path;
- *open(filename)*: opens an existant Document Table;

- *close()*: closes a Document Table; if it was opened with create() then the function saves the collection statistics average document length and closes the file pointer; if it was opened with close() it simply closes the file pointer;
- *insert(entries)*: inserts a vector of entries in the Document Table, it is more efficient because we know the size of the entries and we can write them all at once;
- *getEntryByIndex(index)*: returns the entry at the index passed by parameter, it is the equivalent of the [] operator in a vector.
- *getSize()*: returns the length of the Document Table;
- *getAvgDoclen()*: returns the average document length of the Document Table.

2.4.2 Lexicon

The lexicon implementation follows the one of an external file based hashmap. We truncated the term key to 20 bytes, in order to have a fixed size entry. An entry of the Lexicon has the following format:

<doc_freq, docs_offset, freqs_offset, max_score>

The Lexicon has the following member functions:

- *create(filename, N)*: creates a lexicon at the specified path with N possible hash keys;
- *open(filename)*: opens an existant lexicon;
- *close()*: closes an opened lexicon; if it was opened with create() then the function saves the N value and closes the file pointer; if it was opened with close() it simply closes the file pointer;
- *insert(term, entry)*: inserts an entry in the Lexicon at index *hash(term)*;
- *search(term)*: return the entry corresponding to the term passed as parameter.

2.5 Command line interface

The command line interface (CLI) allows users to index the documents collection, execute queries and create the results file for evaluating the system with trec_eval.

The command **query** allows the user to perform queries after setting the execution mode [*conjunctive_mode, disjunctive_mode, disjunctive_mode_max_score*] and the k relevant documents to return.

The command **index** allows the user to create the index,following the two phase process (parsing and merging) described in the previous sections.

The command **eval** allows the user to load a dataset of queries to be executed and returns as output the *mean response time* of the executions and a file containing the K relevant documents for each query of the dataset. The user needs to specify the query execution mode, the number of relevant documents to return and the file containing the queries. The file is formatted to perform the evaluation of the system with *trec_eval*.

In the following image is shown an example of the CLI execution:

CLI

```
*** Started MSMARCO Search Engine ***
Available commands:
  help - display a list of commands
  query - perform a query
  eval - execute a queries dataset, saving the result file for trec_eval
  index - create the inverted index
  exit - exit the program

Enter a command:
>query
Enter the query execution mode:
    0 : CONJUNCTIVE_MODE
    1 : DISJUNCTIVE_MODE
    2 : DISJUNCTIVE_MODE_MAX_SCORE

>2
Select how many documents return:
>10
Enter the query:

>manhattan project

Results for: "manhattan project"
The elapsed time was 15 milliseconds, 15293700 nanoseconds.

RESULTS:
Doc Id Score
2036644 4.31715
3870080 4.30079
2       4.29498
3615618 4.28213
2395250 4.27013
4404039 4.25136
3607205 4.23599
7243450 4.20026
3689999 4.1146
3870082 4.09159
```

3 Performance

In this section the system performance is shown, all tests have been performed on Windows 11 on a computer that has the following hardware:

- CPU: Intel core i7-1280p
- RAM: 16GB LPDDR4X
- SSD: PCIe 4.0 NVMe 1TB

3.1 Indexing

Indexing performance is an important consideration when developing a search engine and it's the most computationally expensive part.

The indexing time of the system is in the order of 5/6 minutes, using a block dimension value of 2'000'000. More in detail, the parsing phase generally takes 4 minutes and 30 seconds while the merging phase usually takes 1 minute and 30 seconds. At the end of the indexing process, the final compressed inverted index size is:

- inverted_index_docs 292MB
- inverted_index_freqs 205MB
- doc_table.bin 134MB
- lexicon.bin 83MB

3.2 Query Processing

The speed and efficiency with which a search engine can process queries and return results can have a significant impact on the user experience, as well as on the overall performance of the system. Once the indexing stage is completed, the startup phase of the application, it is practically immediate, as it does not need to load any file into memory, being both lexicon and doctable mapped.

The system has beed evaluated using the *queries.dev.small* dataset. Note that we considered as the start time of the query execution the moment in which the lexicon entries are retrieved, or from cache or from the lexicon.

The query processing performance are the following:

- *Query Latency*: **40ms**;
- *Query Throughput*: **25**.

3.3 Dynamic Pruning

In this section the performance of the dynamic pruning implementation with MaxScore are compared to the simple DAAT implementation for disjunctive queries. The query processing performance, evaluated on the *queries.dev.small* daset are the following:

- *Query Latency disjunctive DAAT*: **51ms**;
- *Query Latency disjunctive DAAT + MaxScore*: **40ms**;

3.4 Effectivness Evaluation

For evaluating the performance of the system we use the TREC Eval tool. It was developed by the Text REtrieval Conference (TREC) as a standard way to evaluate and compare the effectiveness of different search engines.

TREC Eval takes as input a set of search results and a set of relevant documents, and it computes a number of evaluation metrics to assess the quality of the search results. These metrics include reciprocal rank, as well as more advanced metrics like mean average precision (MAP) and normalized discounted cumulative gain (NDCG).

In the following tables we compare the results that we obtained testing our search engine against the Terrier search engine using the following queries datasets: *queries.dev.small*, *msmarco-test2020-queries*, *msmarco-test2019-queries*.

Both search engines use BM25 as scoring function, the Porter algorithm for stemming and the same list of stopwords.

dev-small	Metrics			
	RR	nDCG@10	nDCG@100	MAP
MS-Marco SE	0.2023	0.2387	0.2991	0.1990
PyTerrier	0.1962	0.2299	0.2913	0.1929

Table 1: Evaluation on `dev-small` dataset

trec-2019	Metrics			
	RR	nDCG@10	nDCG@100	MAP
MS-Marco SE	0.8262	0.5045	0.5061	0.3062
PyTerrier	0.7950	0.4795	0.5025	0.2864

Table 2: Evaluation on `trec-2019` dataset

trec-2020	Metrics			
	RR	nDCG@10	nDCG@100	MAP
MS-Marco SE	0.8304	0.4957	0.5116	0.3204
PyTerrier	0.8023	0.4936	0.5025	0.2929

Table 3: Evaluation on `trec-2020` dataset

4 Limitations

MSM Search Engine has several limitations as well as possible ways to be improved, both will be described in this section.

Regarding the scoring function, one limitation of the search engine is the fact that we do not take into consideration the order of the query terms and their distance in the documents that contain them, which can be important for determining the relevance of a document. For example, one of the first results of the test query "aziz hashim" is a document that contains the word "aziz" and the word "hashim" at different positions and so it is not a document that concerns with Aziz Hashim. Another limitations of BM25, and consequently of our search engine, is the fact that synonyms or related terms are not handled, as all words are treated as independent: BM25 may not be able to accurately score the relevance of the documents if the search query does not exactly match the words in the documents.

The tokenization employed by the search engine has several limitations: it removes punctuation blindly which may lead to a loss of information about acronyms, hashtags, date or prices; it also does not consider multiword expressions. MSM Search Engine also does not handle spelling mistakes or variations in word forms.

As far as the index compression concerns, one possible improvement is to use *unary encoding* to compress the frequencies instead of variable byte. The frequencies have lots of small values and in such scenarios is highly probable that *unary encoding* performs better than variable byte encoding in both time and space.