University of Pisa

Artificial Intelligence and Data Engineering
Multimedia Information Retrieval and Computer Vision

Search Engine ++
IR system & Index implementation

Dario Pagani 585281
Franceso Londretti 597086
Ricky Marinsalda 585094

December 28, 2023

Contents

Chapter 1

Architecture

1.1 Tokenization

1.1.1 Logic

UTF-8 The MSMARCO corpus contains several wrongly encoded documents, it seems their *spider* wrongly interpreted certain documents as encoded in `latin1` — a eight bit ASCII extension used by old Microsoft's software — instead of `UTF-8` and then converted them in `UTF-8` interpreting their bytes as `latin1` data. To solve this issue we wrote a simple heuristic that tries to detect strange Unicode's entry-points that are likely the result of the processes; and a reverse encoding procedure that converts them back in their original `UTF-8` form. We didn't perform experiments so we cannot tell if this actually improves accuracy on the testing corpus.

Punctuation and splitting Our implementation takes a very simplistic approach to punctuation handling, that is all known western punctuation is removed and replaced with spaces, you can see the punctuation handled by the program in the source file `normalizer/PunctuationRemover.cpp`.

Words, that are tokens, are generated by splitting the resulting text by spaces.

Lower case Changing case of a Unicode word is a non trivial task and there are no fast ready-to-use libraries that directly operate on `UTF-8` — instead they require a conversion to `UTF-16` or `UTF-32`, that we'd rather avoid, — nevertheless we decided not to limit ourselves to only handle the case of ASCII's latin letters but we convert the case of *latin1*'s latin letters — that is a larger subset of Unicode latin letters. In conclusion most words written in latin characters are converted in lower-case.

Stemming Stemming is done using the Porter algorithm.

1.1.2 Libraries

Punctuation To replace punctuation with spaces we make use of regex expressions and run them on Hyperscan (or Vectorscan on non amd64 platforms), that is a fast regex engine that make use of SIMD instructions — such as AVX-512 — to speed up their solution.

Stemming To stem words, we make of the famous Snowball's stemmer implementation.

1.2 Indices' organization

We've designed the system to partition by document, that is we divide the entire document collection in multiple sub-collections, making it possible to parallelize the query execution on multiple machines in a *computing cluster* or on a multi-disk single machine or both.

MapReduce We can think of the query solution problem as a map-reduce problem, as illustrated in Figure 1.1; that is each partition runs on a separated process that generates the top-k documents for its own data, finally a merger procedure generates the global top-k results.

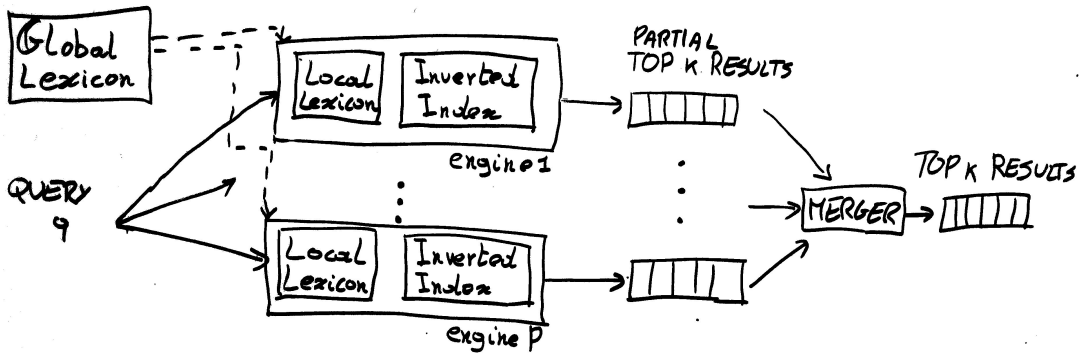

Figure 1.1: Not pictured: each partition has its own document index

1.3 Indices

The inverted indices are relatively straight-forward, they consist of two files; one for the documents' ids and the other for the associated term's frequencies, the former is encoded using *variable bytes* while the latter in *unary code*. The pointers to each region of the index — that is the index's entry relative to a certain term — are stored in the lexicon as explained in section 1.4.2.

1.4 Lexica

Since we're partitioning the dataset by document, to improve its parallelizability we make use of a local lexicon for each partition as well as a global lexicon used to store shared information.

1.4.1 Disk Map

To store the lexica it is necessary to use a disk-based data structure that allows the program to quickly and efficiently seek the datum associated to a particular key, that is an index term. To achieve this we make use of following:

- to reduce disk seek time, the terms' data are compressed with *integer compression* and the terms themselves with *prefix compression*
- to minimize disk accesses, the terms are ordered lexicographically and partitioned in pages of the same size of the disk's page
- terms at the beginning of each page are stored in a separated array that fits in system memory, to quickly locate — through *binary search* or a *trie visit* — in which page a given term is located

Structure Let us call $|B|$ the size of a disk page, $|T|$ the maximum length off all terms stored, m the maximum number of key-value pairs stored in a page, k the number of pages used to store them all, $s_1 > s_2 > s_3 > \dots$ the terms to store and $d_1, d_2, d_3, \dots$ their data; and $s_{b_1} > s_{b_2} > \cdots > s_{b_k}$ the terms that appear at the start of each page.

The following picture shows how our data are organized at high level

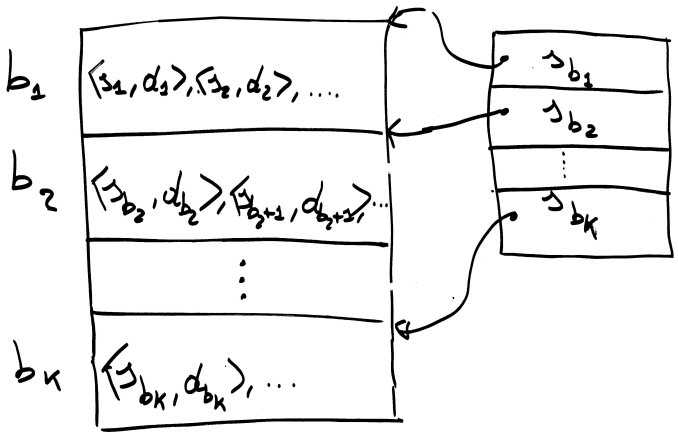

Figure 1.2: Data organization

In the real implementation, the pages' term heads s_{b_i} are only stored in the look-up table, the data d_i are compressed using *variable bytes* and the other strings s_j are compressed by cutting the common prefix they've with the page's head, so a generic key-value pair is more like: $\langle\langle p(s_{b_i}, s_j), s'_j\rangle, c(d_j)\rangle$; where $p()$ is the length of the common prefix, s'_j is the string without the common prefix, and $c(d_j)$ is the compressed datum. The prefix's length is stored on a eight bit unsigned number, thus allowing for a maximum term's length of 255 bytes.

Complexity For simplicity's sake our implementation makes use of binary search to find in which page a term is stored and a linear scan to find it in the page, giving us a time complexity of $O(|T| \cdot \log(k) + m)$, a space complexity, with regard to system memory, of $O(|T| \cdot k)$ — since we only have to keep the heads in memory — and an I/O complexity of $O(1)$, assuming all pages' heads fit in memory.

1.4.2 Data

Local lexica Each local lexicon stores the terms' information as `LexiconValue` during the first pass and then as `SigmaLexiconValue` in the second pass, when the upper bounds are computed and the skipping lists built.

Both structures store the number of docs in which such term appears and pointers to the compressed term frequencies and documents' ids. The latter structure also contains the skipping list and the upper bounds for *TFIDF* and *BM25*.

Global lexicon The only global lexicon just stores the total number of documents each term appear in, this is necessary to correctly compute the *IDF* term.

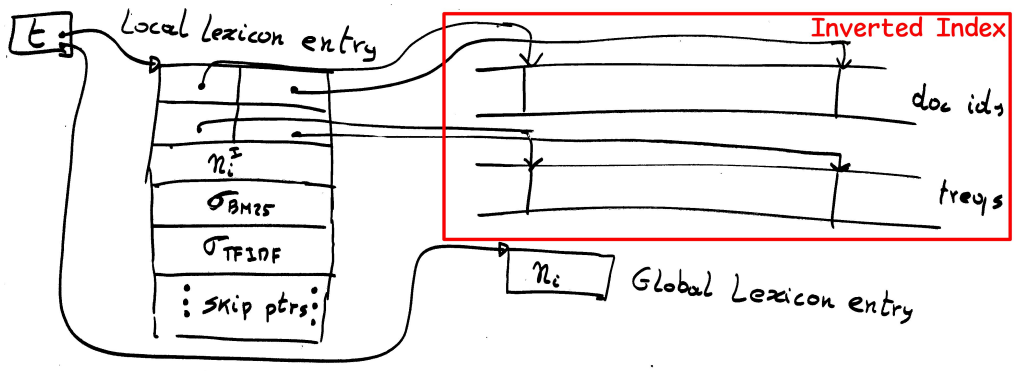

Figure 1.3: Overall representation of the lexica's data. n'_i is an addendum that's used during the building phase to compute the total n_i

1.5 Document Index

The document index is the most straight-forward data structure, all entries have the same length — that is sixteen bytes — allowing access to the document's information in constant time $O(1)$ and with a single I/O read; this is crucial as access to the document's length is continuously performed when ranking with BM25 et similia.

docno To access the document's number we have to pay another I/O access, but that's not very important as the this access is only done once, at the end of the query solution algorithm.

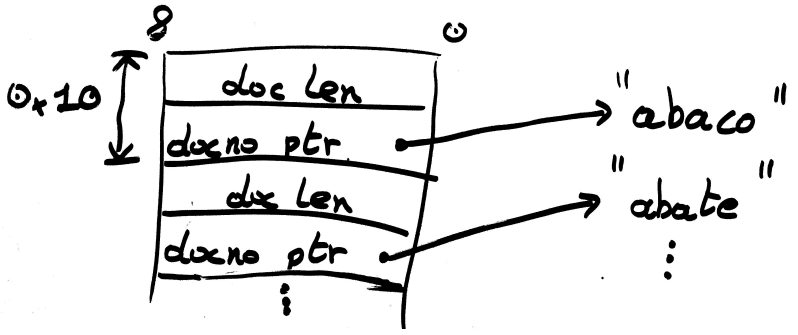

Figure 1.4: The document index, strings are stored at the end of the same file

1.6 Metadata

The metadata file is a sixteen bytes file that contains the collection's metadata that are:

- $|D|$ that is the total number of documents in the collection
- `avgdl` that is the average document length in the collection

1.7 Caches

For simplicity's sake, we do not use caches, even though they might've improved performance; however, since we memory-map all files — through the use of the POSIX system call `mmap()`, — we have some sort of *implicit cache* handled directly by the Kernel, as the most used disk pages are likely to be kept in system memory as the process progresses in its execution.

This is particularly advantageous when accessing the lexica and the document indices; especially for the latter, we can truly exploit this mechanism to obtain real $O(1)$ access times.

Chapter 2

Query Processing

The query processing stage is a critical component of our search engine model. It faces the challenge of efficiently handling sequences of terms within a query. In this chapter, we discuss the implementation choices for query processing, presenting two distinct algorithms: *Document At A Time (DAAT)* and *Block-Max Maxscore (BMM)*.

2.1 DAAT

The *Document At A Time* algorithm is our initial approach to query processing. This algorithm scans the posting lists of query terms in parallel, allowing for conjunctive and disjunctive query processing.

Disjunctive mode The algorithm for disjunctive queries processes a query by scanning posting lists in parallel. The algorithm iterates through the lists, calculating relevance scores for documents, and outputs the final set of matching documents.

Conjunctive mode In a conjunctive query, the algorithm identifies documents that contain all the terms in the query. This involves scanning the posting lists of each term concurrently and identifying documents that appear in all lists. To further optimize the algorithm, we search the document in every posting list parallelly, if it's not present, we can skip its scoring and move to the next document.

2.2 BMM

The *Block-Max Maxscore* algorithm is our second implementation for query processing. This algorithm introduces new data structures saved on the disk to enhance efficiency.

Overview The Block-Max Maxscore (BMM) algorithm processes queries by dividing posting lists into blocks and identifying the maximum score within each block. It avoids scanning the entire posting lists, focusing on blocks likely to contain relevant documents. By efficiently processing blocks, BMM minimizes unnecessary computations, making it more time-effective than Document At A Time (DAAT) for certain scenarios, particularly when dealing with large datasets.

New Data Structures To make the BMM algorithm possible, after the creation of the inverted index, we scan the whole index and calculate some additional values. We call these values *sigmas* and *sigma blocks*. The sigmas are the maximum scores of each posting list, while the sigma blocks are the maximum scores relative to each block of the posting list. Along with the sigmas and sigma blocks, we also save the positions of the start of each block inside a new data structure called `skip_pointers`. These are used to quickly jump to the start of each block, without having to scan the whole posting list.

2.3 Scoring Functions

We decided to use two types of scoring function.

Both formulae use a common term, the *inverted document frequency* — that is $\log_2 \frac{|D|}{n_t}$, — this common term is computed only once and stored to avoid performing the same computation over and over.

Both formulae make use of logarithms, a heavy operation on the *FPU*, to try to speed-up the computation we implemented an integer version of such operation that relies only on the ALU; however, empirically we saw that execution times do not change by much when using implementation and we decided to prefer the floating-point version for greater accuracy.

2.3.1 TFIDF

The first scoring function is the simpler and faster *Term Frequency Inverted Document Frequency (TFIDF)*. Although this scoring function is simpler, it's not really effective in terms of precision value, so it's implemented only for completition purposes.

$$s(q,d) = \sum_{t \in d \cap q} (1 + \log_2(\mathtt{tf}_{t,d})) \cdot \log_2 \left(\frac{|D|}{n_t} \right) \tag{2.1}$$

2.3.2 BM25

The second scoring function is the more sophisticated and widely used *BM25* (Best Matching 25) algorithm. *BM25* takes into account term frequency, document length, and the inverse document frequency, offering a more nuanced measure of document relevance compared to TFIDF.

Parameters The parameters k_1 and b were taken by the following github repository. In this repository, the authors performed a grid search to find the optimal parameters for BM25.

$$s(q,d) = \sum_{t \in d \cap q} \frac{\mathtt{tf}_{t,d}}{k_1 \cdot \left((1-b) + b \frac{\mathtt{dl}_d}{\mathtt{avgdl}} \right) + \mathtt{tf}_{t,d}} \cdot \log_2 \left(\frac{|D|}{n_t} \right) \tag{2.2}$$

We set $k_1 = 0.82$ and $b = 0.68$.

Chapter 3

Benchmarks

3.1 TREC evaluation

To assess the system's performance, we utilized the TREC Eval tool, which was created by the Text REtrieval Conference (TREC) to establish a standardized method for comparing the efficacy of various search engines.

TREC Eval accepts a collection of search outcomes and relevant documents as input, subsequently employing several evaluation metrics to gauge the search result quality. These metrics encompass reciprocal rank, alongside more sophisticated measures such as mean average precision (MAP) and normalized discounted cumulative gain (NDCG).

Results The following table 3.1 makes comparisons of our engine's performance with other search engines freely available, we used the `msmarco-test2020-queries` and set those engines to retrieve the first twenty results for each query.

Metric	**SEPP TFIDF**	**SEPP BM25**	**Anserini BM25**
mAP	0.1189	0.1982	0.1942
RR	0.6527	0.8110	0.8215
ndcg@10	0.3538	0.4750	0.4876
ndcg@20	0.3351	0.4705	0.4705
set P	0.3435	0.4815	0.4667
set R	0.1786	0.2600	0.2496
set F	0.1902	0.2781	0.2670

Table 3.1: comparison of our project and Anserini's

3.2 Time statistics

The table presents the query processing times for the provided collection. All measurements were carried out using a single thread.

Algorithm	**AVG [ms]**	**STD DEV [ms]**	**MAX [ms]**	**[query/s]**
DAAT	25.88	19.69	78.68	51
BMM	6.67	6.29	42.48	149
Anserini				256

Table 3.2: Execution figures for different algorithms, $k = 20$

Furthermore, we have included a histogram illustrating the distribution of execution times for run queries, comparing two methods: BMM and DAAT. This graphical representation allows for a

visual comparison of the execution time distribution between the two query processing approaches.

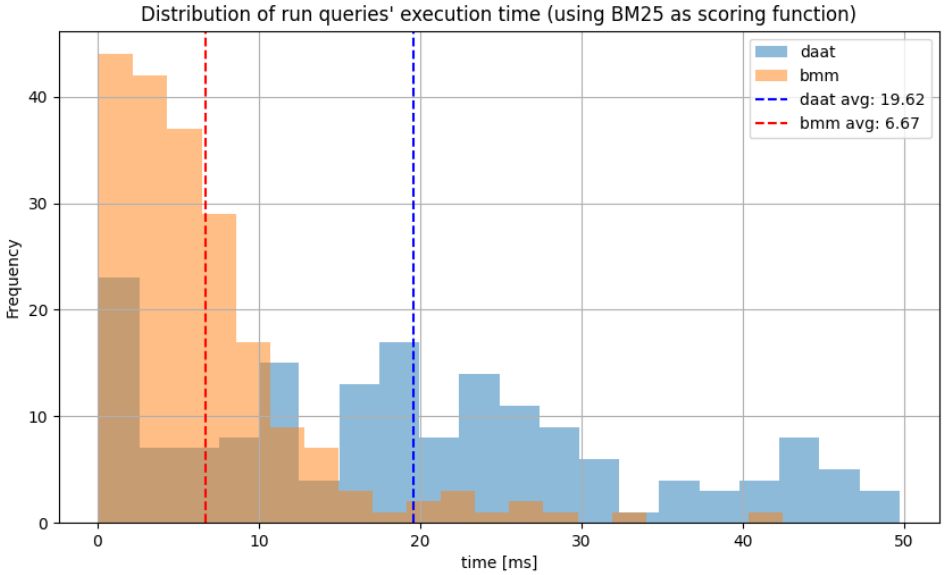

Figure 3.1: Distribution of times

3.2.1 Index's construction time

Listing 3.1: Indexing program's output: it takes less than 1 minute and 20 seconds to build it

```
Init hyperscan db...
building scratch...
building scratch...
building scratch...
building scratch...
Chunk 0 (thread ff) processed in 44.7158s and written in 2.1356s (46s elapsed)
Chunk 1 (thread fe) processed in 47.3445s and written in 2.4206s (49s elapsed)
Chunk 2 (thread fd) processed in 46.4158s and written in 2.4048s (48s elapsed)
Chunk 3 (thread fc) processed in 45.4687s and written in 2.3275s (47s elapsed)
Chunk 4 (thread ff) processed in 14.2876s and written in 0.9657s (15s elapsed)
Indices built in 67.1283s
Built global lexicon from local lexica in 430.707ms
(thread fe) Built skipping list and sigmas for "data/db_0" (max len = 27)
(thread fc) Built skipping list and sigmas for "data/db_2" (max len = 26)
(thread ff) Built skipping list and sigmas for "data/db_3" (max len = 27)
(thread fd) Built skipping list and sigmas for "data/db_1" (max len = 25)
(thread fe) Built skipping list and sigmas for "data/db_4" (max len = 10)
Re-built local lexica in 11.9453s average len = 1.00149
Processed 8841823 documents in 79.5042s 5 indices generated

real 1m19,815s
user 4m16,885s
sys 0m10,968s
```

3.3 Files' size

The following table shows the file sizes presented in megabytes, divided int the indices' main files: Document Index, Local Lexicon, and Posting List.

	Partition 0	Partition 1	Partition 2	Partition 3	Partition 4	Total
Document Index	46	47	46	46	17	**202**
Local Lexicon	22	23	23	23	12	**103**
Posting Lists	131.5	174.5	174.5	173.5	61.7	**715.7**
Global Lexicon						**14**
TOTAL						**1034.7**

Table 3.3: Files' sizes, figures in megabytes

To provide a clearer visualization, we have included a tree diagram below where the files' structure is depicted.

Listing 3.2: Files tree

```
[4.0K] .
|-- [4.0K] db_0
| |-- [ 46M] document_index
| |-- [ 22M] lexicon
| |-- [ 14M] lexicon_temp
| |-- [124M] posting_lists_docids
| `-- [7.5M] posting_lists_freqs
|-- [4.0K] db_1
| |-- [ 47M] document_index
| |-- [ 23M] lexicon
| |-- [ 14M] lexicon_temp
| |-- [167M] posting_lists_docids
| `-- [7.5M] posting_lists_freqs
|-- [4.0K] db_2
| |-- [ 46M] document_index
| |-- [ 23M] lexicon
| |-- [ 14M] lexicon_temp
| |-- [167M] posting_lists_docids
| `-- [7.5M] posting_lists_freqs
|-- [4.0K] db_3
| |-- [ 46M] document_index
| |-- [ 23M] lexicon
| |-- [ 14M] lexicon_temp
| |-- [166M] posting_lists_docids
| `-- [7.5M] posting_lists_freqs
|-- [4.0K] db_4
| |-- [ 17M] document_index
| |-- [ 12M] lexicon
| |-- [7.2M] lexicon_temp
| |-- [ 59M] posting_lists_docids
| `-- [2.7M] posting_lists_freqs
|-- [ 14M] global_lexicon
`-- [ 16] metadata
```

3.4 Limitations

SEPP has some drawbacks. One issue is with our word normalization algorithm, our search engine's use of it doesn't handle synonyms or related terms well. Basically, every word is treated separately, so if your search doesn't exactly match the words in the documents, BM25 might not score them accurately for relevance.

Our search engine's tokenizer also has its problems, as it blindly removes all punctuation, which means we might lose important stuff like acronyms, hashtags, dates or prices. Furthermore, it doesn't recognize phrases made up of multiple words. On top of that, SEPP doesn't catch spelling mistakes or different word forms.

Possible solutions SEPP's drawbacks include word normalization issues — enhance it for synonyms and related terms. Modify text tokenization to retain punctuation for acronyms, hashtags, and multiword expressions. Implement spell-check and word form recognition for improved flexibility. Utilize NLP, user feedback, and BM25 adjustments for more accurate search outcomes. For example: experiment with BM25 parameters to optimize the search engine's scoring mechanism. Fine-tuning these parameters could improve relevance scoring, even when search queries don't precisely match document words.