UNIVERSITÀ DI PISA

IR SEARCH ENGINE

Project for Multimedia Information Retrieval course

A.Y. 2022/2023
Artificial Intelligence and Data Engineering
Stefano Bianchettin, Matteo Mugnai, Filippo Puccini

1. SOMMARIO

1. SOMMARIO 2

2. INTRODUCTION 3

3. MAIN MODULES 4

3.1 Build structures 4

3.1.1 Compression strategies 5

3.2 Query Processing 6

3.2.1 Cache implementation 6

3.3 Performance Test 7

3.4 Common 7

4 PERFORMANCES AND STATISTICS 8

4.1 EFFICIENCY 8

4.2 Cache 8

4.3 Effectiveness 9

4.4 Stemming and stopword removal 9

5 LIMITATIONS 10

2. INTRODUCTION

This project aims to create a sample search engine for text retrieval and document processing for academic purposes. The program can index a collection of 8.8 M documents and to process user queries over it.

Project's source code is available on *Github* at the following link:

Buch97/MIRCV_Search_Engine (github.com)

The whole project has been developed in *Java* language and it is composed of three sequential steps:

- Indexing of the collection
- Query processing
- Performance tests

3. MAIN MODULES

The project is divided into four different modules:

- **Build structures**: this module is dedicated to collection indexing. At the end it generates an inverted index, a document index and a lexicon. All structures are stored on disk in binary format.
- **Query processing**: it handles the entire flow of query processing; it receives a textual query as input and generate as output a ranking of documents in order of relevance according to a score function.
- **Performance test**: this module evaluates the performances of the system on a query dataset built in a suitable format.
- **Common**: it contains *Java* classes and utility functions that all other modules need during their execution.

3.1 BUILD STRUCTURES

In this module we read and process the entire collection document by document.

The first stage is a pre-processing phase. It is necessary to eliminate all malformed characters or lines of the original file. Once applied all transformations to the text, it's possible to perform a space-based tokenization to all documents. A single token is composed by the term, the document id (of the one in which the term occurs) and how many times it occurs. All generated tokens are stored in a memory block. A block saves tokens until a specific threshold (the 80% of JVM memory is consumed) is crossed. At this point we need to build a data structure to index each block. This operation is performed through ***Single pass in-memory indexing*** algorithm. When all collection has been processed, we obtain several separate blocks of tokens with a partial vocabulary for each one. The last step of this module is a merge phase in which we aggregate all partial blocks into a single one, obtaining the final inverted index and the lexicon. The document index is completed before the merge phase, when the full scan of the collection is done.

In pre-processing phase, we applied the following transformations:

- HTML tags and URLs removal
- Non-ASCII characters removal
- Additional whitespaces deletion
- Punctuation removal
- Upper camel case split
- String > 64 char removal

Stopwords removal has been implemented (using this list) by pruning all terms present in a textual file containing common English stopwords.

Stemming has been implemented following ***Porter Stemmer*** algorithm for *Java*.

The program provides the possibility to enable/disable those two steps through a compile flag.

This module can be run also in debug mode by setting up a specific compile flag. In this alternative execution the whole process is performed on a sample collection with 1000 documents. This possibility is useful for testing purposes to check if the program works properly.

All data structures have been stored on disk in binary format through the usage of ***Java File Channel*** class. Moreover, inverted index has been stored in a compressed format in order to reduce space wasting for allocation, to speed up read operation and to store some posting lists in main memory during *Query Processing* execution.

The indexing of the whole collection and the merging phase have a duration of 1 hour and half.

The following table shows the memory space occupied by all necessary structures.

DATA STRUCTURE	MEMORY OCCUPED
Document Index	**636** Mb
Lexicon	**128.22** Mb
Inverted Index: Doc Id (compressed)	**268.47** Mb
Inverted Index: Term Frequencies (compressed)	**32.46** Mb

3.1.1 Compression strategies

Two different encoding algorithms have been used for compression:

- **Gaps + Variable Byte**: applied to sorted *Docids* in all postings. This approach has greater memory footprint than Gamma or Delta encodings but is much faster in decompression phase thanks to the fact that all codewords are byte aligned. To reduce the memory footprint, we compress the gaps between docids. We can exploit this representation because Docids are large integers in most of the cases.
- **Unary**: applied to *Term frequencies* in all postings. This choice is driven by the fact that term frequencies are very often small numbers (1,2…) because in a document a term appear a limited number of times since all stopwords have been removed. We must pay something if in a document a term is very popular, but we gain every time we encounter few occurrences (only one bit is used to store 1 as natural number). Since in Java can read or write no less than a single byte, we minimise the wasting space compressing in a Bitset Java representation all the term frequencies of the posting list relative to the term and convert in a sequence of bytes. In the worst case only the last byte of the sequence can have seven bits wasted.

3.2 QUERY PROCESSING

This module is dedicated to receiving queries from users, process posting lists and return a ranked list of relevant documents according to the submitted query.

The first stage includes some operations that will be useful in subsequent phases, like loading lexicon in memory. Next, the program parses the query, applies pre-processing to query terms and then it must retrieve and decompress all posting lists involved in the task; these operations are performed by a thread for each query term, to make these tasks in parallel. After this we need to pass through a scoring algorithm: ***Document at a time*** or ***Max Score***. These algorithms can be run either in conjunctive mode or in disjunctive one. The scoring functions implemented are: ***BM25*** and ***TF-IDF***. After scanning all postings and computing final scores the program prints a list of results (docid + score) sorted in decreasing order by score.

Also, this module allows to the user the possibility to set some compile flags. He/she can choose how many results want to see (10 or 20), the scoring function to use (BM25 or TF-IDF) and the kind of query (conjunctive or disjunctive).

3.2.1 Cache implementation

To implement the cache, we used the *Guava* library provided by *Google*, loading the already decompressed posting lists into the cache. In the query processing phase, after text-processing, we go to build the posting lists of the terms within the query, reading inside the inverted_index_doc_id and inverted_index_doc_frequency files: if the term with its posting list is already in the cache we can avoid the reads and decompression of the Docids and term frequencies, otherwise the full procedure is performed and eventually the posting list of the newly requested term is loaded into the cache; the data structure of the cache is the HashMap, with term as key and posting lists as value (array of posting objects).

The policy by which evictions are executed is LRU (last recently used) and is managed internally by the *Guava* library. Also, to give a limit to the size of the cache, we gave a weight to each record, based on the length of the posting list, so that cache elements are deleted if the total weight of the records goes beyond a certain threshold. This gave us some problems, as we saw, after several tests, that the cache filled up too quickly than expected, even by increasing the size considerably. This is probably due to the fact that through text-processing, particularly stemming, the length of the posting lists (on average) becomes very high and only a few terms are loaded into cache. One possible solution is to stop using the posting lists to calculate the maximum size of the cache, and to use *Guava's softvalues* method, which encloses each item within the cache in a soft reference: those are deleted only if there is a memory request from the system. However too many soft references may affect the system performance, as the garbage collector is called too often. For these reasons, we decided to run the tests by disabling the cache and create a test specifically for it, shown in Section 4.2.

3.3 PERFORMANCE TEST

This module estimates the efficiency and effectiveness of our system. For the first aspect a performance test is run over a query collection available on msmarco GitHub repository. The program performs a test over this collection processing queries one by one. At the end final results are shown to the user: average time elapsed, cache hit rate, fastest query and slowest one. In this way we can get trustable results and make some considerations about efficiency. We can run different kind of tests by changing some flag at running time.

For what concern effectiveness, we conduct also additional tests based on trec eval: to do this, we exploit this tool, https://trec.nist.gov/trec_eval/ which allowed us to obtain various metrics, such as mean average precision or p@10. The document to be submitted to the tool is prepared during the test phase, by setting a specific flag which enables the creation of the file in the format indicated on the msmarco website.

3.4 COMMON

This module shares classes and methods with other modules. In package beans are defined:

- **Lexicon Entry**:
  - Term (String): 64 bytes
  - Doc_frequency (int): 4 bytes
  - Coll_frequency (int): 4 bytes
  - Offset_doc_id_start (long): 8 bytes
  - Offset_doc_id_start (long): 8 bytes
  - Offset_tern_freq_start (long): 8 bytes
  - Offset_tern_freq_end (long): 8 bytes
  - TermUpperBound (float): 4 bytes

A total size of 108 byte for a single entry in Lexicon binary file. For simplicity and efficiency, we decide to load offline the entire lexicon in memory without the need to maintain a data structure of reference of the lexicon entries.

- **DocumentIndex Entry**:
  - DocId (int): 4 bytes
  - DocNo (String): 64 bytes
  - Doc_len (int): 4 bytes

A total size of 72 byte for a single entry in DocumentIndex binary file, mapped in memory using a Java MappedByteBuffer.

- **InvertedList**: the posting lists of the term are loaded in memory in a list of Posting object and maintain the current position of the posting processed during the Query Process. This object can be converted in a Java iterator list.

In package FileChannelInvIndex the class handle the file channels for inverted index: doc_index and term_frequencies_index, map them in memory using the MappedByteBuffer for the entire duration of query process and get the data stored in the files.

4 PERFORMANCES AND STATISTICS

Below there is an overview of main tests performed. All these tests are performed using the msmarco *queries.dev.tsv* and *qrels.dev.tsv* (only for trec eval).

4.1 EFFICIENCY

SCORING ALGORITHM	QUERY MODE	SCORING FUNCTION	AVG RESPONSE TIME
DAAT	Disjunctive	TF-IDF	35 ms
		BM25	50 ms
	Conjunctive	TF-IDF	9 ms
		BM25	9 ms
MaxScore	Disjunctive	TF-IDF	19 ms
		BM25	37 ms
	Conjunctive	TF-IDF	8 ms
		BM25	8 ms

We performed a comprehensive test regarding the execution times of the DAAT and MaxScore algorithms, using both TFIDF and BM25 as scoring functions, in disjunctive and conjunctive modes. As can be seen, in the case of disjunctive queries, MaxScore is almost twice as fast as DAAT, both using TFIDF and BM25; the latter is slower since we need to perform a read on the document index to get the document length. In the case of conjunctive queries, the performances are equivalent, and, in both cases, we have a low average response time.

4.2 CACHE

FULL TEXT PREPROCESSING	CACHE ENABLED	CACHE DISABLED
Enabled	20 ms	19 ms
Disabled	308 ms	400 ms

As mentioned, a caching mechanism of the decompressed posting lists has been implemented. The benchmarks were performed using Max Score and TF IDF as scoring function, analysing the case in which terms are pre-processed or not.

If we do not perform pre-processing of terms (in particular stemming and stopwords removal), we have a good improvement in performance, since the cache has a relatively high hit rate (around 38%), because, without the stemming phase, the posting lists are on average much shorter and therefore a greater number of terms are loaded into the cache (considered the eviction policy explained before). If we perform pre-processing, however, we have a very slight deterioration due to a low cache hit rate (around 5%) and the length of the posting lists, which mean that only a few terms are loaded into the cache.

4.3 EFFECTIVENESS

QUERY MODE	MAP	P@10
TF-IDF	0.1293	0.0272
BM25	0.1832	0.0388

The third test we performed was the calculation of the effectiveness measures, in particular the map and the p@10, of the TFIDF and BM25 scoring functions, using MaxScore as the scoring algorithm. As it is possible to see BM25 has slightly better results, however compensated by the greater slowness shown previously.

4.4 STEMMING AND STOPWORD REMOVAL

FULL TEXT PROCESSING	DAAT	MaxScore
Enabled	35 ms	19 ms
Disabled	1041 ms	400 ms

The last test concerns the performance obtained by our search engine when stemming and stopwords removal is performed. The test was performed using both MaxScore and DAAT as scoring algorithms and TFIDF as scoring function, showing the mean response time. As you can see, disabling word pre-processing significantly increases the average response time, also causing a significant waste of space, since the lexicon and inverted index must index many more terms.

5 LIMITATIONS

The previous paragraph stresses the fact that performances of our system are quite satisfactory for academic purpose, although it is possible to improve different aspects in terms of coding and implementation. The main aspect that requires more attention or need to be revised:

- **Lexicon stored in memory is not the best solution**: in a web search engine this solution can't work because lexicon dimension grows very fast in proportion of number of documents in collection. Google web search indexes 130 trillion of pages against only 8.8 million of documents in this project. If we obtain a size of 105 Mb with this collection, it's not possible to store lexicon in memory if we think to Google environment.

- **Doc Index and lexicon have not been compressed on disk**: in order to save memory space and computational time it would be better to apply a compression algorithm also to document index and lexicon. This can make a big difference in a commercial Web search engine.

- **Posting lists are fully loaded in memory**: it is better to load dynamically portions of posting lists instead of the whole list to prevent possible memory space problems.

- **Cache preload has not been implemented**: a cache preload would be better to speed up also first queries during query processing execution. The preload can be done in a smart way by considering most frequent term present in popular queries asked by user. Since we adopt a term caching strategy, we can store posting lists (or a portion of them) related to popular terms in main memory thanks to preload.

- **Static pruning techniques are not applied**: we didn't exploit static pruning in this project, but it would be very useful in reducing the amount of posting lists (especially if are unnecessary) and consequently to speed up scoring process.

- **64-byte length for lexicon terms could be not sufficient**: we assumed for simplicity that all string greater than 64 chars come out from tokenization errors and malformed documents, so we prune them to obtain a fixed size for each vocabulary entry. In a real environment we could meet very long term in documents (even > 64) with a meaning. German language for example has some very long meaningful words. Since the indexing has been done over an English collection of documents, we made this assumption.