University of Pisa

Multimedia information retrieval and computer vision

Project

Search engine

Bruni Davide

A.Y. 2023-2024

The code is available on GitHub at:
`https://github.com/DavideBruni/MIRCV_project`

Contents

Chapter 1

Introduction

The text search engine software is implemented in Java using the JDK19 and it can operate in two modes:

- inverted index creation
- query processing

Each mode can be executed by launching the appropriate jar with parameters that will be shown below.

1.1 Index creation

The project directory contains the ***indexer.jar*** file. In order to create an inverted index run:

```
java -jar indexer.jar path/to/collection parse compression [path/to/log/dir]
```

How to set parameters:

- **path/to/collection**: a *string* representing the path to the collection containing data to be indexed.
  Note: the collection file must be compressed as a '.tar.gz' file with a unique tsv file named 'collection.tsv' inside.
- **parse**: a *boolean*, set it to true in order to perform stopwords removal and stemming.
- **compress**: a *boolean*, set it to true in order to compress the inverted index.
- [**path/to/log/dir**] **(Optional)**: an optional path referred to the directory where the log file will be stored. If not provided, error logs will be printed on stderr.

1.2 Query processing

The project directory contains the ***query_processor.jar*** file. In order to perform a query or generate a file useful for trec evaluation run:

```
java -jar query_processor.jar parse compression score_standard evaluation k
```

How to set parameters:

- **parse**: a *boolean*, set it to true if the index was created using the parse parameter set to true.
- **compress**: a *boolean*, set it to true if the index was created using the compression parameter set to true.
- **score_standard**: a *string*, only BM25 and TFIDF are accepted as value. If the parameter doesn't match either standard, TFIDF will be used.
- **evaluation**: a *boolean*, if true, the 'msmarco-test2020-queries.tsv' file will be used, and a file valid for trec evaluation will be generated. If false, a command-line interface will be shown waiting for a query. Once a query is submitted, the top k pair *(docno,score)* will be shown.
  It is possible to execute queries in both disjunctive and conjunctive mode (for conjunctive mode it is necessary to prefix the '+' character before the query).
- **k**: an *integer*, it's the maximum number of documents that the system returns in response to a query (i.e., the top k documents that best match the query).

Chapter 2

How the system works

2.1 Index creation

The system reads the file supplied as an input parameter using the **TarArchiveInputStream and GzipCompressorInputStream** classes and it searches for the file containing the collection to be indexed. The collection is indexed using the **SPIMI algorithm**, merging partial results into a single InvertedIndex.
For each document in the collection, the software extracts the document's PID and the text:

- Since the collection is composed of documents with monotonically incremental PIDs starting from 0, the **DocID is equal to PID+1**.
- The text is tokenized and preprocessed.

2.1.1 Text preprocessing

The function *getTokens* of the **unipi.aide.mircv.parsing.Parser** class perform the following operation on the provided text:

- Replaces the HTML tags and punctuation with a white space.
- Brings the text to lower case.
- Discards words encoded with more than 64 Byte [1]
- Splits the tokens based on white spaces.
- For each token:
  - Discard if the token contains invalid UTF-8 char
  - For non-numeric characters, if there are more than two consecutive identical characters, it replaces them with only two consecutive identical characters (it is assumed that if there are more than two consecutive identical characters in a token, it is an error)

[1] 64 is a default value that can be changed

- If parse flag was set to true it performs stop-words removal and applies PorterStemmer [2]

Example of preprocessing:

"University-of-PissSa" *becomes* "university of pisa"

2.1.2 Posting lists creation

A mapping between token and a posting list is kept in memory and once we get a list of token representing a document, for each token we add to the posting list related to it the documentId and the term frequencies of that term in that document.[3].
Note that a single (uncompressed) posting list is implemented as an **Arraylist of documentIds (integer) and a Arraylist of frequencies (integer)**. Term frequency and documentId are linked by the index position.
Once the 80% of the allocated memory for the JVM is reached, the (partial) index is written on disk.

2.1.3 Writing on disk

Before writing the index to disk, we sort the posting lists in a lexicographic ascending order (each posting list is already sort by documentId in ascending order by construction). Then we write the Lexicon and the posting list in the following file:

- **Lexicon** in rootDir[4]/temp/lexicon/partN.dat.
  Notes on writing:
  - Tokens are always written using 64 bytes (if the token is smaller, trailing white spaces are used as padding) in order to facilitate the retrieval when using binary search in the query processing part.
  - Lexicon is always written uncompressed.
  - During SPIMI and merge phase different information are written to disk (in order to minimize the memory footprint).
    * During the "SPIMI phase" token, documentFrequency, offsetInDocumentIdFile, offsetInFrequencyFile, maxDocId are written (80 bytes for each entry are used).
    * In "merge phase" are written the same things (expect for maxDocId) and docIdsLength, frequencyLength, BM25TermUpperBound and TFIDFTermUpperBound are added (100 bytes for each entry are used).
- **PostingLists**:
  - **Document ids** in rootDir/temp/docIds/partN.dat.

[2]https://github.com/caarmen/porter-stemmer/tree/master
[3]When we see one term for the first time, we have to create a new posting list
[4]The default root dir is data/invertedIndex

– **Frequencies** in rootDir/temp/frequencies/partN.dat

Note on PostingList writing:

– if compression is chosen, posting lists are written compressed also when partial index are written.

– Before writing the posting list document ids, 2 integer representing the **maxDocId** and the **number of postings** are written.

– Before writing the posting list frequencies, an integer representing the **number of byte** occupies by the frequencies is written.

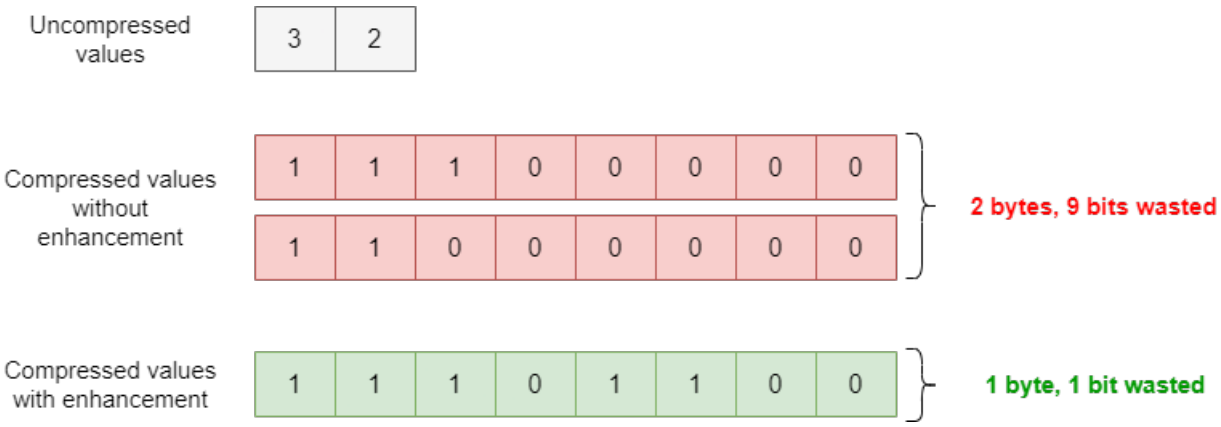
The service information are added for two reasons: the write function is the same used in the Merge phase when skipping pointers are used and because they are necessary especially when decompress a posting list (in a file generated by SPIMI or in a block when skipping pointers are used).

Then the DocumentIndex is written on file. Note that the DocumentIndex is a simple ArrayList where the index correspond to the pid of the document and the value is the documentLength.

2.1.4 Compression

Only the postingLists are compressed:

- Document ids are compressed using **Elias-Fano** compression.
- Term frequencies are compressed using **Unary compression**.
  In this version of Unary compression a number is represented as: as many number of 1s as the value of the number itself followed by one 0.
  Since in Java it's possible to write to disk only using the byte as the smallest unit, the fastest way to implement compression would be to always write a byte, even if the compression required only one bit (leading to a waste of 7 bits for each number in the worst case), so I decided to make the compression slightly slower in order to save more space, representing the compressed number on the exact number of bits required (avoiding waste and thus making a small "enhancement")



2.1.5 Merge

Premise: in the merge phase it was decided to divide the posting lists into blocks if necessary: a **threshold** of 2048 bytes was chosen. If the size of the posting list exceeds this threshold, then it will be divided into blocks of size $\sqrt{\text{document frequency}}$.
After having written all the partial indexes we move on to the merging phase. For each partial Lexicon, a stream is opened, one entry per partition at time is kept in memory and the smallest token (considering the lexicographic order) within all lexicons is chosen. Then for each partial index that contains the chosen term, the posting list is read, but **in order to avoid the loading of the complete uncompressed posting list** in memory the following action are performed:

- The posting list of the first partition containing that token that has not been read yet is loaded into memory and decompressed if necessary.
- If the size of the posting list loaded in memory is greater than the threshold:
  - Write on disk as many block as possible (i.e. size of posting list loaded / block size). Since this division will not be exact, the leftovers are kept in memory (they will be written in the next block).
  - Calculate term upper bound for this posting list part.
- Else accumulates postings until the threshold is reached or until the entire posting list has been analyzed.
- Then write on disk last block (and calculate term upper bound) and the lexicon entry.

Once all posting lists have been processed, the collection statistics (which include total documents lengths and number of document in the collection) are written and the temporary files are deleted.

2.2 Query processing

Once a query is received the same pre-processing applied to the documents (see paragraph 2.1.1) is performed. Then tokens are search on the Lexicon (stored on disk) performing a binary search. For each token, lexicon entries and posting list are loaded into memory.
Then the MaxScore dynamic pruning algorithm is executed in order to return the best k documents.
Note: if the score standard chosen is BM25, the document index is loaded in memory too.

Chapter 3

Evaluation

All evaluation are performed on the MSMARCO Passages collection.

3.1 Index creation evaluation

In this section I want to compare the performance of the Index creation in terms of storage space and execution time based on whether the posting lists are compressed or not and whether it is pre-processed or not.

Index	DocIds	Frequencies	Lexicon	Document Index	Colleciotn Statistics	Tot.
Parsed and compressed	370 MB	59 MB	118 MB	34 MB	20 B	581 MB
Parsed and not compressed	692 MB	674 MB	118 MB	34 MB	20 B	1518 MB
Not Parsed and compressed	685 MB	115 MB	137 MB	34 MB	20 B	971 MB
Not Parsed and not compressed	1362 MB	1341 MB	137 MB	34 MB	20 B	2874 MB

Table 3.1: Index files size

Index	Time (mm:ss:ms)	Milliseconds
Parsed and compressed	37:11:638	2231638
Parsed and not compressed	39:42:702	2382702
Not Parsed and compressed	50:04:014	3004014
Not Parsed and not compressed	42:03:217	2523217

Table 3.2: Index time creation

As excepted, the best storage situation is when the posting lists are compressed (since there is almost 1GB difference between index create with or without compression) and when the pre-processing is fully performed since there are less token to be stored in the lexicon (and related postings list to be stored).

3.2 Query processing evaluation

In this section I want to compare the performance of the query processing when the index is created performing stopwords removal and stemming or not and when the score standard chosen is TFIDF or BM25. In order to perform these comparison I used:

- the TREC DL 2020 queries
- the TREC DL 2020 qrels
- MAP@20 and nDCG@20 as metrics

Stopwords removal and stemming	Score Standard	MAP@20	NDCG@20	Avg time per query
True	TFIDF	0.1813	0.3084	42 ms
True	BM25	0.1903	0.3234	44 ms
False	TFIDF	0.1618	0.3051	342 ms
False	BM25	0.1750	0.3160	713 ms

Table 3.3: trec_eval results and average execution per query

3.3 Comments

As we can see from both evaluation, the best situation is when the stopwords are removed and stemming and compression are performed, because we have lower storage space used, a better average response time per query and better values of bot MAP and nDCG.
About the **compression ratio of parsed index, its value is 2.92.**
About the score standard used, since the average response time per query difference is negligible, BM25 returns better results.

3.4 Limitation and future implementation

The most significant limitation is probably associated with the lexicon: to save space, it could be compressed and the storage of 64 bytes for each word could be avoided by implementing a dynamic length for token storage. Additionally, it could be saved a single term upper bound between BM25 and TFIDF, which will then be used during query processing.
A future implementation related to speed up query results response time is the implementation of caching strategies like keeping in memory query results or posting lists.

Chapter 4

Major functions and modules

In this chapter the most important class and modules are reported, for each directory is present a README.md file explaining classes and methods present inside the directory.

4.1 Compression

- Class **unipi.aide.mircv.model.Bits**.
  It's an auxiliary class useful for EliasFano and Unary compression. Main methods:
  - **static int readUnary(final byte[] in, long bitOffset)**
  - **static long writeUnary(final byte[] in, long bitOffset, int val)**
  - **static int readBinary(final byte[] in, long bitOffset, int numbits)**
  - **static void writeBinary(final byte[] in, long bitOffset, int val, int numbits)**
- Class **unipi.aide.mircv.model.EliasFano**.
  This class implement the EliasFano logic compression, using the Bits class methods. Main methods:
  - **public static void compress(final List<Integer> in, byte [] out, int l, long highBitsOffset)**;
    This method split high and low bits, then for each group of low bits written, update the counter of the cluster with the same high bits. When the cluster changes, it write the unary value of the counter of the last cluster(s) paying attention to empty clusters. **The compressed list is written in the out array concatenating lowbits | highbits**. If the last byte of lowbits has remaining unused bits, they are wasted since highbits starts always in a new byte, however in the worst case only 7 bits are wasted) .
  - **public static List<Integer> decompress(final byte[] in, final int length, final int maxDocId)**;
    This method reads how many number have the same prefix (say n numbers), than reads n low bits and retrieve the original uncompressed number. Then increments the prefix and starts again until all bits are read.

- **public static int get(final byte[] in, final int maxDocId, final int length, final int idx, EliasFanoCache cache)**
  This method get the idx value of the compressed list. Lowbits are read simply reading the idx*l input array index. High bits instead are retrieved summing up unary parts until idx value is reached. For speed up reading, I use an EliasFanoCache instance where the last high bits offset, number of docIds and last high part are saved.

4.2 Posting Lists

- Class **unipi.aide.mircv.model.PostingLists**.
  This class has as attribute a **Map<String, PostingList>** in order to associate token to posting list during SPIMI algorithm. Main methods:

  - **public void sort()**.
    This method sort the Map parsing it to a LinkedHashMap.
  - **public void add(int docId, String token, int frequency)**.
    This method add docId and frequency to the posting list associated to the token given as parameters. If there's not a posting list for that token, a new one is created.

- Class **unipi.aide.mircv.model.PostingList**.
  This **abstract** class when is used during Index Creation have all method implemented by CompressedPostingList or UncompressedPostingList and there's no relevant attribute. Instead when is used for query processing, two main attributes are:

  - **BlockDescriptor**
    It contains information about the block currently analized, like maxDocId, numberOfPostings, end index of docIds and frequency block.
  - **LexiconEntry**
    Reference to the LexiconEntry in order to be fast when document frequency is necessary during scoring.

  About documentIds and termFrequencies: they are both represented as two array of integer in the class **unipi.aide.mircv.model.CompressedPostingList**, instead they are represented as two ArrayList<Integer> in the class **unipi.aide.mircv.model.UncompressedPostingList**.
  About methods: **next(), nextGEQ(int id) and id()** are implemented in classes **unipi.aide.mircv.model.CompressedPostingList** and **unipi.aide.mircv.model.UncompressedPostingList** following this logic:

  - The complete posting list is loaded in memory and saved in two array of byte[] (List of Integer if uncompressed), one per docIds and one for frequencies.
  - For the first block, BlockDescriptor is created

- When function next() or nextGeq() needs to read from a new block, the BlockDescriptor is updated by replacing it with a new instance. This new instance is created by reading information from the two arrays of bytes.
- The indexes related to the current position in the posting list are reset.

4.3 Lexicon and LexiconEntry

Inside the Class **unipi.aide.mircv.model.Lexicon**, the most important attribute is **Map<String,LexiconEntry> entries** which map an entry to a token. Each LexiconEntry is formed by the following attributes:

- **int df**; the document frequencies of the term
- **int docIdOffset**; the starting offset of the document ids list on disk.
- **int frequencyOffset**; the starting offset of the term frequencies list on disk.
- **int docIdLength**; used to load the compressed posting list in memory, since the block length is variable.
- **int frequencyLength**; used to load the compressed posting list in memory, since the block length is variable.
- **int maxDocId**; this is attribute is used only during SPIMI and merge operations. In the final lexicon is not present.
- **double BM25_termUpperBound**; used for dynamic pruning in order to avoid the calculation in the online phase
- **double TFIDF_termUpperBound**; used for dynamic pruning in order to avoid the calculation in the online phase

4.4 Score and MaxScore

Scoring function for single term (used during query processing) and to calculate term upper bounds (both TFIDF and BM25 are lways calculated when the index is created) are implemented here: **unipi.aide.mircv.queryProcessor.Scorer**. In the same class is also implemented the method **PriorityQueue<DocScorePair> maxScore(PostingList[] postingLists, boolean conjunctiveQuery)** where the DocScorePair is a class used to associate the score to a document and the boolean parameter indicate if the query (and so the MaxScore algorithm) must be executed in a conjunctive mode or not.