UNIVERSITÀ DI PISA

Information Retrieval System implementation

Multimedia Information Retrieval and Computer vision

Master's Degree in Artificial Intelligence and Data Engineering

University of Pisa

Massagli Lorenzo

Patimo Antonio

Table of Contents

Introduction

Information retrieval is a field of study that focuses on how to organize, search for, and retrieve information efficiently and effectively. A common way for people to search for information is by using a search engine. To facilitate search, there are certain data structures called indexes that are designed to work with large amounts of data. The goal of this project is to create an index based on a specific collection of documents called the Passage ranking dataset and use it to efficiently search through many documents. The project will involve designing and implementing data structures, creating a scalable index, and developing the ability to process queries. The implementation of this project will be done using the Java programming language.

Passage Ranking Dataset: https://microsoft.github.io/msmarco/TREC-Deep-Learning-2020

Github Project Repository: https://github.com/EhiSuper/Information-Retrieval-Project

Architecture

The architecture of an information retrieval (IR) system is made up of two main parts: the indexing component and the search component. The indexing component takes the collection of documents and prepares it in a way that allows for quick retrieval of information. The search component receives queries from users, who are looking for specific information, and uses the index to find the most relevant documents. These documents are then ranked based on their relevance to the query.

Index

The index data structure is an essential part of an information retrieval system.
It is made up of:

- Inverted index, which is a mapping of terms to the frequency of those terms within documents.
- Lexicon, which contains statistics about terms across the entire collection.
- Document index, which has information about individual documents and their statistics.
- Collection statistics, which provide overall statistics about the collection.

Document Collection

For the purpose of this project, the document collection known as the Passage ranking dataset, which can be found at the following web page: https://microsoft.github.io/msmarco/TREC-Deep-Learning-2020, was selected as the source of data. This collection consists of approximately 8.8 million documents, totalling around 2.2GB in size.
Each document in the collection is represented by a single line of text, using the following format:

{docno}\t{content}\n

where "{docno}" represents a unique identifier for the document and "{content}" represents the actual content of the document. The document number and content are separated by a tab character, and the line is terminated by a newline character.

Index Construction

The process of building the index for the information retrieval system involves two options for the format of the index: a binary format or an ASCII format (used for debugging purposes). The index construction can be specified by using a specific compile flag. The Indexer component will use the ZipInputStream class to decompress the collection.tar.gz file, which contains the document collection. This allows the documents to be parsed as they are being uncompressed. To handle text written in non-ASCII character sets, the Unicode standard of representation has been adopted and UTF-8 encoding has been selected as the scheme to use.

Text Pre-processing

To ensure that the index does not include empty pages, the length of the content of each document is calculated and checked to make sure it is non-zero. Any malformed characters that are encountered are handled using the BufferedReader class, which throws a MalformedInputException. This exception is caught, and the default operation is to replace the malformed character. The text of each document is then converted to lowercase and all punctuation and non-ASCII characters are removed using the regular expression "[^a-zA-Z0-9]". If the appropriate flags are set to true, the process of removing stop words and applying stemming (using the Snowball English stemmer) is carried out on the text. A posting is created for each term, if it does not already exist, and the document identifier is added to the term's posting list. After all the documents have been processed, the collection statistics are updated to include the number of documents and the average length of a document. These statistics will be used by the ranking algorithm later.

Memory Management

The process of constructing the index for the information retrieval system is based on the Single-pass In-memory indexing (SPIMI) algorithm and is performed in the createIndex() function. This approach allows the index to be built in a scalable manner, with intermediate results being stored on disk and considering any hardware constraints that may be present. Before each document is processed, a check is performed to see if the amount of memory being used is greater than or equal to a certain percentage (in our tests, we used 75%) of the available memory. If this threshold is reached, the partial data structures are written to disk, using the writeBlock() function, and the memory is cleared by creating new objects for the data structures and calling the System.gc() function to remove any unused objects. This ensures that the index construction process does not exceed the available memory resources. Each time the memory threshold is hit, the partial data structures will be stored in the disk, specifying in the file name the block identifier. The following file system structure will be used to store our partial data structures:

```
Output
 ├── DocIds
 │    ├── docIds0.dat
 │    ├── …
 │    └── docIdsN.dat
 ├── Frequencies
 │    ├── freq0.dat
 │    ├── …
 │    └── freqN.dat
 ├── Lexicon
 │    ├── lexicon0.txt
 │    ├── …
 │    └── lexiconN.txt
 └── DocumentIndex
      ├── documentIndex0.dat
      ├── …
      └── documentIndexN.dat
```

After indexing all documents present in the collection, the files containing the partial data structures are merged. The final file system structure will be the following:

```
Output
 ├── CollectionStatistics
 │    └── collectionStatistics.txt
 │
 ├── DocIds
 │    └── docIds.dat
 │
 ├── Frequencies
 │    └── freq.dat
 │
 ├── Lexicon
 │    └── lexicon.txt
 │
 │ ── DocumentIndex
 │    └── documentIndex.dat
 │
 └──Skipping
      |── skiPointers.dat
      └── lastDocIds.dat
```

The merging phase in manly done in the mergeBlocks() function present in the index class. As described in the SPIMI algorithm a pointer to every block file is opened using a reader buffer and a write buffer is opened for every output file. Selecting every time, the minimum term in the pointed lexicon blocks, it is performed the merging over the document identifiers and term frequencies files. This is possible thanks to the fact that the document identifiers are ordered and separated on every block file. During the merging phase based on the decided length of the posting list block the skipping pointers and the relative document identifiers are saved to two different files to implement skipping in the query processing phase. Indeed, in the lexicon other than the offset of the docIds and freq files are also saved the offset of these two files.

Integer Compression

To decrease the index size, we implemented the Variable byte code compression algorithm and we used to compress both document identifiers and frequencies. We also compressed the document index file, the skipPointers file and the lastDocIds file. The indexing program has the second parameter that can be "text" or "bytes" to write files in ASCII format during debugging or in binary format for performance. The algorithm is implemented in the VariableByteCode class that extends the Compressor interface present in the indexing package.

Query Processing

The query processing accepts a range of flags that specify how it should operate.
These flags include:

1) an integer value K, which determines the number of relevant documents that should be returned as a result
2) a string that can be either "tfidf" or "bm25" and specifies which scoring function to use
3) a string that can be either "daat" or "maxscore" and specifies how the posting lists should be processed
4) a string that can be either "conjunctive" or "disjunctive" and specifies how the queries should be processed
5) a Boolean value (true or false) that specifies whether stopword removal is activated
6) a Boolean value (true or false) that specifies whether stemming is activated

Once the query processor is started, the system will accept queries from the user and return the docno of the top K relevant documents (according to the query) based on the selected scoring function. After the results are returned, the program will wait for the next query to be input.

Posting list iterators

The posting lists of the query tokens are accessed by a custom posting list iterator and offers the following operations:

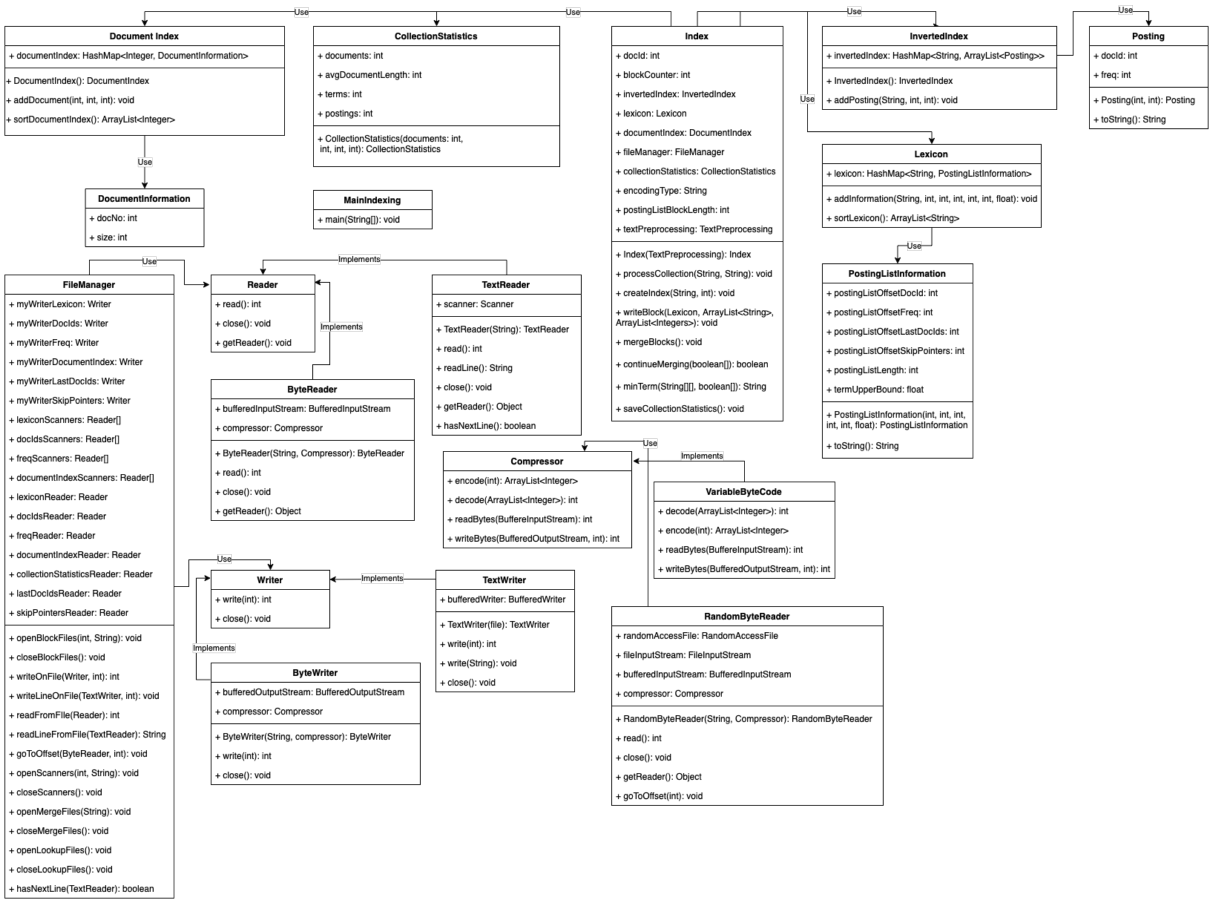
- docID() returns the docid of the current posting
- score () returns the score of the current posting
- next () moves sequentially the iterator to the next posting
- nextGEQ(d) advances the iterator forward to the next posting with a document identifier greater than or equal to d
- isFinished() returns a Boolean that specify if the posting list has been completely visited or not.

Dynamic Pruning

To improve the efficiency of query processing, a dynamic pruning strategy that utilizes the MaxScore algorithm has been implemented. This strategy helps to avoid the need to score documents that will not be able to be included in the final top K results. The pruning strategy uses several components, including document upper bounds, term upper bounds, a min-heap, thresholds, and essential/non-essential posting lists, to achieve this goal. These components work together to enable the system to quickly identify and eliminate irrelevant documents, allowing it to focus on the most promising candidates and speed up the overall query processing process.

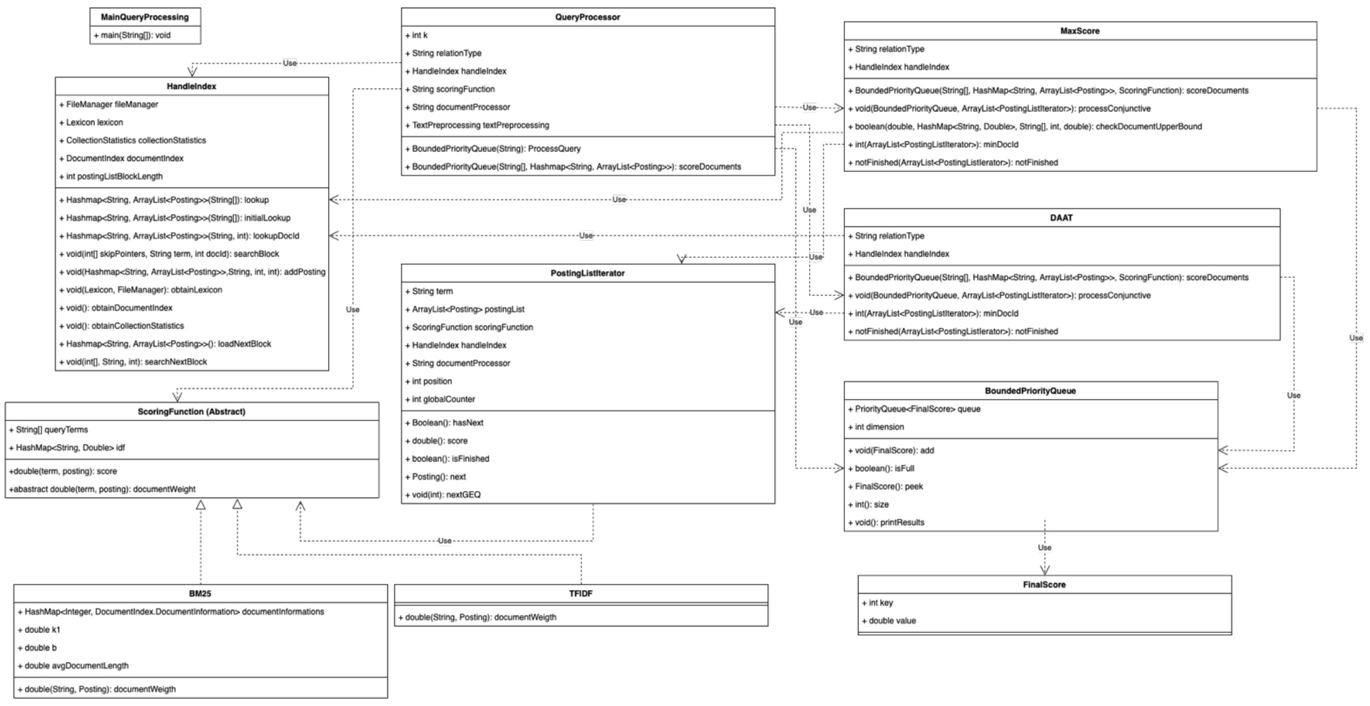
Java Class Diagram

Indexing class diagram



Use
Use
Use
Use
Document Index
+ documentIndex: HashMap<Integer, DocumentInformation>
+ DocumentIndex(): DocumentIndex
+ addDocument(int, int, int): void
+ sortDocumentIndex(): ArrayList<Integer>
CollectionStatistics
+ documents: int
+ avgDocumentLength: int
+ terms: int
+ postings: int
+ CollectionStatistics(documents: int, int, int, int): CollectionStatistics
Index
+ docId: int
+ blockCounter: int
+ invertedIndex: InvertedIndex
+ lexicon: Lexicon
+ documentIndex: DocumentIndex
+ fileManager: FileManager
+ collectionStatistics: CollectionStatistics
+ encodingType: String
+ postingListBlockLength: int
+ textPreprocessing: TextPreprocessing
+ Index(TextPreprocessing): Index
+ processCollection(String, String): void
+ createIndex(String, int): void
+ writeBlock(Lexicon, ArrayList<String>, ArrayList<Integers>): void
+ mergeBlocks(): void
+ continueMerging(boolean[]): boolean
+ minTerm(String[][], boolean[]): String
+ saveCollectionStatistics(): void
InvertedIndex
+ invertedIndex: HashMap<String, ArrayList<Posting>>
+ InvertedIndex(): InvertedIndex
+ addPosting(String, int, int): void
Posting
+ docId: int
+ freq: int
+ Posting(int, int): Posting
+ toString(): String
Lexicon
+ lexicon: HashMap<String, PostingListInformation>
+ addInformation(String, int, int, int, int, int, float): void
+ sortLexicon(): ArrayList<String>
PostingListInformation
+ postingListOffsetDocId: int
+ postingListOffsetFreq: int
+ postingListOffsetLastDocIds: int
+ postingListOffsetSkipPointers: int
+ postingListLength: int
+ termUpperBound: float
+ PostingListInformation(int, int, int, int, int, float): PostingListInformation
+ toString(): String
DocumentInformation
+ docNo: int
+ size: int
MainIndexing
+ main(String[]): void
Implements
Implements
FileManager
+ myWriterLexicon: Writer
+ myWriterDocIds: Writer
+ myWriterFreq: Writer
+ myWriterDocumentIndex: Writer
+ myWriterLastDocIds: Writer
+ myWriterSkipPointers: Writer
+ lexiconScanners: Reader[]
+ docIdsScanners: Reader[]
+ freqScanners: Reader[]
+ documentIndexScanners: Reader[]
+ lexiconReader: Reader
+ docIdsReader: Reader
+ freqReader: Reader
+ documentIndexReader: Reader
+ collectionStatisticsReader: Reader
+ lastDocIdsReader: Reader
+ skipPointersReader: Reader
+ openBlockFiles(int, String): void
+ closeBlockFiles(): void
+ writeOnFile(Writer, int): int
+ writeLineOnFile(TextWriter, int): void
+ readFromFIle(Reader): int
+ readLineFromFile(TextReader): String
+ goToOffset(ByteReader, int): void
+ openScanners(int, String): void
+ closeScanners(): void
+ openMergeFiles(String): void
+ closeMergeFiles(): void
+ openLookupFiles(): void
+ closeLookupFiles(): void
+ hasNextLine(TextReader): boolean
Reader
+ read(): int
+ close(): void
+ getReader(): void
ByteReader
+ bufferedInputStream: BufferedInputStream
+ compressor: Compressor
+ ByteReader(String, Compressor): ByteReader
+ read(): int
+ close(): void
+ getReader(): Object
TextReader
+ scanner: Scanner
+ TextReader(String): TextReader
+ read(): int
+ readLine(): String
+ close(): void
+ getReader(): Object
+ hasNextLine(): boolean
Use
Implements
Compressor
+ encode(int): ArrayList<Integer>
+ decode(ArrayList<Integer>): int
+ readBytes(BuffereInputStream): int
+ writeBytes(BufferedOutputStream, int): int
VariableByteCode
+ decode(ArrayList<Integer>): int
+ encode(int): ArrayList<Integer>
+ readBytes(BuffereInputStream): int
+ writeBytes(BufferedOutputStream, int): int
Use
Implements
Writer
+ write(int): int
+ close(): void
TextWriter
+ bufferedWriter: BufferedWriter
+ TextWriter(file): TextWriter
+ write(int): int
+ write(String): void
+ close(): void
Implements
ByteWriter
+ bufferedOutputStream: BufferedOutputStream
+ compressor: Compressor
+ ByteWriter(String, compressor): ByteWriter
+ write(int): int
+ close(): void
RandomByteReader
+ randomAccessFile: RandomAccessFile
+ fileInputStream: FileInputStream
+ bufferedInputStream: BufferedInputStream
+ compressor: Compressor
+ RandomByteReader(String, Compressor): RandomByteReader
+ read(): int
+ close(): void
+ getReader(): Object
+ goToOffset(int): void

Query Processing Classes


MainQueryProcessing
+ main(String[]): void
QueryProcessor
+ int k
+ String relationType
+ HandleIndex handleIndex
+ String scoringFunction
+ String documentProcessor
+ TextPreprocessing textPreprocessing
+ BoundedPriorityQueue(String): ProcessQuery
+ BoundedPriorityQueue(String[], Hashmap<String, ArrayList<Posting>>): scoreDocuments
MaxScore
+ String relationType
+ HandleIndex handleIndex
+ BoundedPriorityQueue(String[], HashMap<String, ArrayList<Posting>>, ScoringFunction): scoreDocuments
+ void(BoundedPriorityQueue, ArrayList<PostingListIterator>): processConjunctive
+ boolean(double, HashMap<String, Double>, String[], int, double): checkDocumentUpperBound
+ int(ArrayList<PostingListIterator>): minDocId
+ notFinished(ArrayList<PostingListIerator>): notFinished
HandleIndex
+ FileManager fileManager
+ Lexicon lexicon
+ CollectionStatistics collectionStatistics
+ DocumentIndex documentIndex
+ int postingListBlockLength
+ Hashmap<String, ArrayList<Posting>>(String[]): lookup
+ Hashmap<String, ArrayList<Posting>>(String[]): initialLookup
+ Hashmap<String, ArrayList<Posting>>(String, int): lookupDocId
+ void(int[] skipPointers, String term, int docId): searchBlock
+ void(Hashmap<String, ArrayList<Posting>>,String, int, int): addPosting
+ void(Lexicon, FileManager): obtainLexicon
+ void(): obtainDocumentIndex
+ void(): obtainCollectionStatistics
+ Hashmap<String, ArrayList<Posting>>(): loadNextBlock
+ void(int[], String, int): searchNextBlock
DAAT
+ String relationType
+ HandleIndex handleIndex
+ BoundedPriorityQueue(String[], HashMap<String, ArrayList<Posting>>, ScoringFunction): scoreDocuments
+ void(BoundedPriorityQueue, ArrayList<PostingListIterator>): processConjunctive
+ int(ArrayList<PostingListIterator>): minDocId
+ notFinished(ArrayList<PostingListIerator>): notFinished
PostingListIterator
+ String term
+ ArrayList<Posting> postingList
+ ScoringFunction scoringFunction
+ HandleIndex handleIndex
+ String documentProcessor
+ int position
+ int globalCounter
+ Boolean(): hasNext
+ double(): score
+ boolean(): isFinished
+ Posting(): next
+ void(int): nextGEQ
BoundedPriorityQueue
+ PriorityQueue<FinalScore> queue
+ int dimension
+ void(FinalScore): add
+ boolean(): isFull
+ FinalScore(): peek
+ int(): size
+ void(): printResults
ScoringFunction (Abstract)
+ String[] queryTerms
+ HashMap<String, Double> idf
+double(term, posting): score
+abastract double(term, posting): documentWeight
BM25
+ HashMap<Integer, DocumentIndex.DocumentInformation> documentInformations
+ double k1
+ double b
+ double avgDocumentLength
+ double(String, Posting): documentWeigth
TFIDF
+ double(String, Posting): documentWeigth
FinalScore
+ int key
+ double value
Use

IR evaluation

Trec_Eval

Trec eval is a widely used tool in the TREC (Text REtrieval Conference) community for evaluating the performance of an ad hoc retrieval system. It is given a results file and a standard set of judged results, and it produces a series of evaluation metrics that can be used to compare the system's performance to other systems. In our project, we used the files queries.dev.small and qrels.dev.small, which can be found at the following web page: https://microsoft.github.io/msmarco/TREC-Deep-Learning-2020, to evaluate the performance of our solution against the Terrier search engine. The results of this evaluation are shown below:

Trec_eval metrics (BM25)		
Metric	Terrier	Our solution
mAP	0.1929	0.1738
RR	0.1962	0.1776
nDCG@10	0.2290	0.2227

Time statistics

In the table above are shown the time statistics about the most important tasks of the search engine. The statistics are show in milliseconds and minutes. The query used are the same used with trec_eval.

Tasks	Time in milliseconds	Time in minutes
Indexing using byte encoding with compression	443159ms	7.5m
Indexing using text encoding	490599ms	8.0m
Test queries (DAAT)	247862ms	4.13m
Average time per test query (DAAT)	35ms	

Index size Statistics

In the table above we can observe the results in term of components size of the final index. It is shown the difference between the byte encoding using compression and text encoding. As we can expect the byte encoding using compression is much more space efficient, considering that the index files occupy about the half of the space. The lexicon remains more or less of the same space because it is always text encoded.

Documents	Byte encoding with compression	Text encoding
Lexicon	72.6 MB	70.2 MB
Document identifiers	742.5 MB	1.55 GB
Frequencies	197.3 MB	394.8 MB
Document index	75.3 MB	165.7 MB
Skip pointers	15.2 MB	33.2 MB
Posting list blocks delimiter	6.8 MB	13.9 MB

Conclusions and Limitations

The solution proposed can be an effective solution for an index structure based on the mentioned collection. The following limitations have been highlighted:

- We used variable-byte encoding for frequencies and docIDs to achieve a higher efficiency but to have a higher compression rate it's possible use a different compression scheme for the frequencies, for example unary.
- As we use not very long posting lists in general, the skipping is not performing as well as could perform in a more complex scenario (as we can load all the postings related to a term in memory and the overhead of lookups is higher).
- We precomputed only the term upper bounds using the TFIDF scoring function, so the max score algorithm can be used only with that scoring. It is possible to add the precomputed term upper bound for the BM25 scoring function to use also that function with the max score algorithm.