Search Engine Implementation

Mehran Ali Banka

November 15, 2023

Abstract

The purpose of this project is to : 1) Create an efficient inverted index, lexicon and page table. 2) Create a search engine on top of that which supports conjunctive and disjunctive query processing with reasonable efficiency. The corpus used is located at corpus. The indexer should be able to index the 22GB corpus with a few hundred MBs of RAM using disk based sorting, and store a compressed version back on disk. The lexicon and page table can be handled completely in memory and should be saved on disk as well for retrieval later on. We should be able to load the posting of a particular term from the compressed index back into main memory in constant time.

1 Introduction

Creating an efficient architecture to store inverted indexes and access them in memory is crucial to implement a good information retrieval system. In this project, I took a 22GB corpus in .trec format, and stored its inverted index in compressed form on disk. For the purpose of compression, I used varByte encoding. I created an *IndexEntry.java* object to represent the posting of a term. In this class, I have implement various efficient methods to deal with this problem which I will discuss below. This class stores postings in a fixed block size (*no of docs per block*), which can be controlled with a parameter. This helps with reducing the size of each block by storing the differences of the increasing docIds in the block, instead of actual docIds. The page table is built on the go, as the docIds are processed. At the end, the posting file is compressed using varByte and the lexicon is updated to store the *file pointer offset* of each posting, and the *max score of the term.* This lexicon can get very big in size, so it is periodically written to disk and flushed during the compression process. This behaviour can also be controlled using a parameter *lexicon flush limit* After the index is created, the engine should support conjunctive and disjunctive query execution. It should be possible to load only the page table and the lexicon in memory, and read the postings from disk and uncompress them in memory. Various optimizations have been done to make execution faster which will be discussed below.

2 Methods

2.1 Index Building Stage

IndexEntry

To handle the postings of a term in memory efficiently, we need a good data structure to encapsulate various complicated operations. I created a data structure *IndexEntry.java* for this purpose. This class represents the complete inverted index of a term in-memory, at any given time, till the term is pushed to disk. It has the following attributes:

IndexEntry.blockSize: Represents the size of each block in the inverted index. By default, this is set to 128, but can be changed.

IndexEntry.noOfDocs: Represents the total number of documents in the inverted index of the term. This is used to update the lexicon during compression.

IndexEntry.noOfBlocks: Represents the total number of blocks in the inverted index of a term. This is useful for merging two inverted indexes of the same term.

IndexEntry.docIds: This is a list of blocks, each of size blockSize, except the last one. Each block contains docIds in increasing order.

IndexEntry.frequencies: This is a list of blocks, each of size blockSize, except the last one. Each block contains frequencies corresponding to the docId.

IndexEntry.java has the following methods to handle postings in memory efficiently:

IndexEntry.toFileFormat(): Take a IndexEntry object and returns a string representation of how it would be stored in a file. For example, if a IndexEntry has 2 blocks of size 2, with some given docIds and frequencies, then this method would return $\boxed{\text{2 2 3 d1 d2 f1 f2 d3 f3}}$. Here 2,2,3 represent block size, no of blocks, and no of docs respectively. d1, d2 is the first block of size 2. d3 is the second block, of size 1.

IndexEntry.createFromLine(line): Takes a String representation of a posting, and converts it into a IndexEntry Object with appropriate block sizes. This is the reverse operation of the above method.

IndexEntry.mergePosting(IndexEntry other): Merges another IndexEntry with this IndexEntry. For example, if for a block size of 2, we have these postings for the same term: $\boxed{\text{2 2 3 1 5 2 1 7 3}}$. and $\boxed{\text{2 1 2 7 9 1 4}}$, then this would merge them to $\boxed{\text{2 2 4 1 5 2 1 7 9 4 4}}$

IndexEntry.reduceDocIDSize(): Reduces the docIds in each block by storing the difference from the previous docId in the block. This improves the quality of compression a lot.

IndexEntry.computeImpactScores(): Computes the max-score of a particular posting which is then stored in the lexicon. This helps in optimizing disjunctive query processing using the max-score algorithm.

Compressor

I have written a *IndexCompressor.java* class to do a varByte encoding of the postings and store it as binary data in the final compressed index. During compression, the lexicon is updated with the offset of a particular term posting, as well the number of documents containing that term (N). The offset helps in getting the posting back in memory in constant time, and N helps in scoring. The class also contains a method to decode a varByte compressed line of binary data from the index file.

Workflow

The diagram below explains the logical overview of the code. It can be summarized as follows:

1) Read the main corpus N bytes at a time, where N is controlled with a parameter, *part index file size.* For the N bytes read into the buffer, create a in-memory intermediate index, update the page table, and at the end, store the index of in a temp location. The index must be stored in a lexicographically sorted order on disk.

2) The above process would create baseline index files of the order of S/N, where S is the size of the corpus. Start merging these index files, two at a time, and store the newly merged indexes in a new location, deleting the older ones. Continue to do so till we get a single merged index. The number of merge iterations would be of the order of *log(S/N).* During the merge operations, it is important to merge files in a lexicographic order to make sure the order of the postings is maintained. If the two terms coming from the two files are different, then we store the posting of the smaller term and update its file pointer. If the terms are the same, then we use *IndexEntry.mergePosting()* to merge them efficiently and store the posting. We are equivalently performing mergeSort on the two index files. Log the page table to the disk. In this project, I have not compressed the page table.

3) Start the varByte compression on the final merged index, and update the lexicon (flushing it periodically)

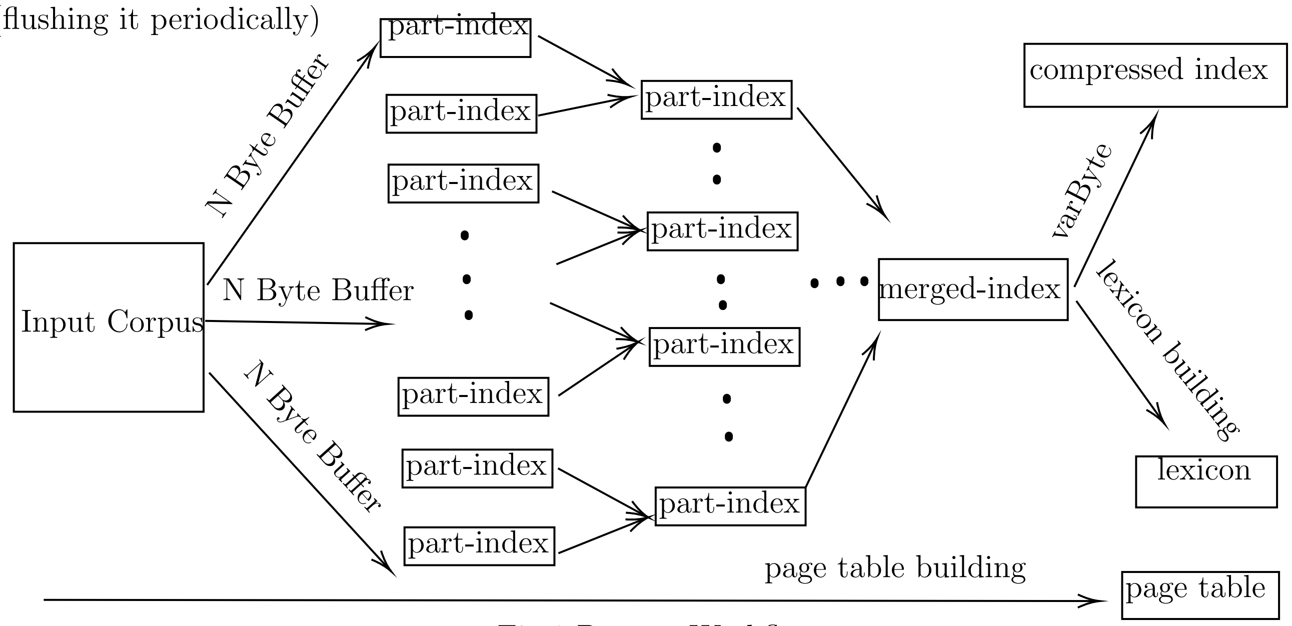

Fig.1 Process Workflow

2.2 Query Processing Stage

Changes to the Index Structures

During the implementation of Query Processing, I made some changes to the index structures for efficiency.

1) In the page table, I stored the document length as well as the starting offset of the document in the main corpus. The document length is needed for computing the BM25 score of the document, and the offset of the document is needed to generate a snippet efficiently. I have created a separate class, *PageTableEntry.java* to hold the page table values in memory.

2) In the lexicon, I stored the maxScore of a term in the lexicon entry as well, along with the previously held offset in the index file. The maxScore is needed for running the MaxScore DAAT algorithm, in disjunctive query processing.

3) In the Inverted Index, I stored the terms by running them through a regex cleaner, *SEHelper.java*, in order to have a more accurate lexicographic representation of the corpus. This involves removing stop words, punctuations, etc. During snippet generation, I still use the unedited corpus, as that blends more suitably with the snippet generation task.

I did not run the entire code again to achieve this, I instead created a separate class, *TaskRunner.java* to work on top of the existing structures.

Actual Query Processing

After the index structures were modified, I created the following classes to handle the actual search implementation:

QueryProcessor

This is the main class that gets called in order to run the search engine. It starts by loaded the page-table and the lexicon from the disk into the main-memory, and then starts taking the user inputs.

ResultEntry

This is an object to hold one document from the results of a particular search. It contains the document ID, the document URL, the snippet, and the BM25 score.

IndexEntryLocation

This is an Iterator class for IndexEntry. It stores the pointer to the IndexEntry that it is iterating over, the index of the block which it points to right now, and the index of the document inside the block that it points to right now. Some simple helper methods have been written - nextDocID() returns the next documentID in the IndexEntry, next() moves the iterator by one location, reset() resets the iterators block index and doc index back to zero.

QueryProcessorDetail

This class handles the actual implementation of the conjunctive and disjunctive query processing. Before calling the actual logic for either one, the query is tokenized, passed through *SEHelper.java*, and the index for the terms is loaded from the disk to main memory. These indexes are stored as *IndexEntry.java* objects. After that, *IndexEntryLocation* objects are created as iterators on the *IndexEntry.java* objects for efficient iteration over the documents. The following methods hide the underlying implementation of crucial iteration operations:

openList(query term): Loads the index entry of the term from the disk by reading the offset in the lexicon. Creates the *IndexEntryLocation* iterator and returns it.

nextGEQ(IndexEntryLocation, docID): Goes into the passed IndexEntryLocation and returns the first document ID that it finds which is greater than or equal to the passed docID. It returns a *MAXDID* if the list is exhausted. Since the documents in the *IndexEntry* are stored in a sorted manner on 128 sized blocks, this method can efficiently jump over blocks that are not of interest and only iterate over the block whose last docID is less than the passed docID.

openList(IndexEntryLocation): Destroys the IndexEntryLocation object to free up memory, once processing is done

getScore(IndexEntryLocation): Returns the BM25 score of a document to which the IndexEntryLocation iterator is pointing at this time.

Conjunctive Query Processing steps:

1) Call openList() on all the query terms and create the Index Iterators

2) Sort the iterators (*IndexEntryLocation*) in increasing order of the number of documents that they contain. This is very useful for making large jumps.

3) Start with the first document of the smallest iterator, and continue checking for its existence in other iterators using the nextGEQ() method. If all documents contain it, score the document and save it in the heap. Increment the current docID by one. If all the documents don't contain it, continue with the last docID returned by the nextGEQ() function and try again.

4) Take the top 10 results from the heap, create *ResultEntry* objects on them, and return for output.

Disjunctive Query Processing steps:

1) Call openList() on all the query terms and create the Index Iterators

2) Sort the iterators (*IndexEntryLocation*) in decreasing order of their maxScore.

3) Build a continuous sum array holding the sum of the maxScore of all iterators so far. Determine a value (*theta*) as the threshold (For this project, I kept it as the highest maxScore amongst the individual iterators). Divide the iterators into two lists - Essential and Non-Essential. The Essential list contains the iterators which can sum up to have a score greater than theta. We do a full scan of these lists as each document is of interest. The Non-Essential lists contain all the rest of the iterators. We do a lookup into these lists using the nextGEQ() and reset() methods. The lookup is for the documents returned by the Essential lists. Store the top-10 results in the heap.

4) Take the top 10 results from the heap, create *ResultEntry* objects on them, and return for output.

SnippetGeneratorNaive

For this Project, I created a naive snippet generation class. Once all the top-K documents have been finalized, if (*Constants.snippet-gen*) is set to true, this class returns a dynamic snippet of the documents for a given query. It tokenizes the query, reads the complete document using its offset location in the page table, and gets the top-3 lines from the document as snippets. The top-3 lines are determined by the frequency of the terms in the query in those lines.

3 Results

This is the format of the final structures:

Index posting: S B N d1 d2 ... dS f1 f2 .. fS d11 d12 .. d1S f11 f12 .. f1S ... where S = block size, B = no of blocks, N = no of docs. The difference of docIds in stored in the document block, not the actual docID, before compressing.

Lexicon entry: file-offset —— byte-length —— maxScore

Page Table entry: page-URL —— doc-length —— doc-offset-in-main-corpus

In this project, I only compressed the inverted index, not the lexicon and page-table. These are the final sizes I got:

Compressed Inverted Index Size:

– With VarByte and delta encoding - 3.46 GB

– With only VarByte - 5.13 GB

Lexicon Size: 851 MB

Page Table Size: 276 MB

Table 1: Query Execution Statistics (milli-seconds)

		Snippet — No-Snippet	
One Word Query	Conjunctive	23	34
	Disjunctive	30	39
Two Word Query	Conjunctive	120	145
	Disjunctive	147	168
Three Word Query	Conjunctive	390	424
	Disjunctive	510	569
Four Word Query	Conjunctive	870	950
	Disjunctive	1223	1290
Five Word Query	Conjunctive	1102	1220
	Disjunctive	1652	1784

4 Discussion

The model did relatively well. The results were pertinent, and most queries run in under a second. Snippet generation is not as expensive as I thought. Using the DAAT and maxScore gives significant performance gains.

4.1 Future work

These are the next areas of improvement in this project:

1) Document Expansion for better retrieval.
2) Smart Snippet Generation Algorithms.
3) Index tiering for performance gains.
4) Experiment with tokenizers and stemmers.
5) Use of more complex models than conjunctive and disjunctive.
6) Caching most frequently searched terms.