Compressed Inverted Index Creation

Mehran Ali Banka

October 24, 2023

Abstract

The purpose of this project is to create an efficient inverted index, lexicon and page table. 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 *no of docs containing 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*

2 Methods

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.

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.

3) Log the page table to the disk. In this project, I have not compressed the page table.

4) Start the varByte compression on the final merged index, and update the lexicon with the file pointer offsets and number of documents for the term. If the lexicon reaches the limit set by a parameter, flush the lexicon to disk.

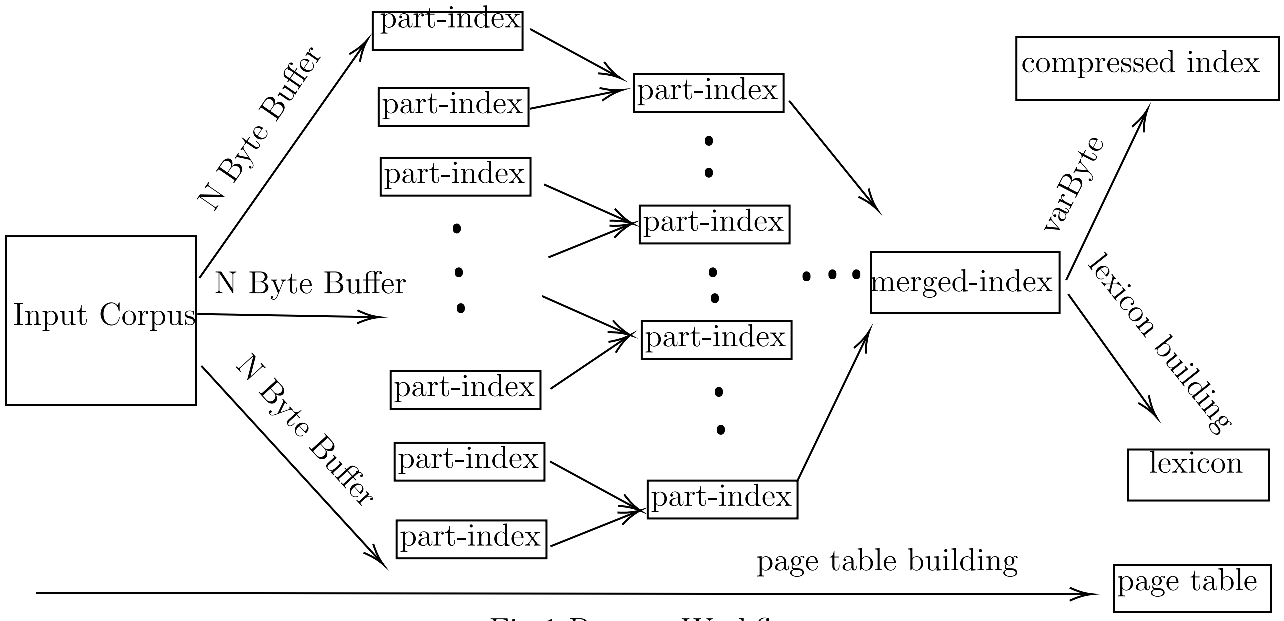

Fig.1 Process Workflow

3 Results

This is the format of the final structures:

Index posting: $\boxed{\text{S B N d1 d2 ... dS f1 f2 .. fS d11 d12 .. d1S f11 f12 .. f1S ...}}$ 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: $\boxed{\text{file-offset no-of-documents}}$

Page Table entry: $\boxed{\text{page-URL}}$

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: 3.99 GB

Lexicon Size: 1.90 GB

Page Table Size: 225 MB

4 Discussion

4.1 Future work

These are the next areas of improvement in this project:

1) Compress page table and lexicon as well.

2) Use a better Tokenizer. I experimented with *StanfordNLPTokenizer* but I did not implement it in this project. This will create a better-quality inverted index as well as reduce the size of the lexicon.

3) Use a stemmer. I did not use a stemmer in this project, although I did experiment with a stemmer from opennlp library. This can be explored again.

4) Use more file buffers for merge. In this project, I merged files two at a time. I was concerned about the computing resources at hand. In the future, I can increase that number and/or use Multithreading. This will bring down the time it takes to merge the final index.