COL 764 Assignment 1 - Inverted Index Construction

Navya Jain

2019CH10106

1 Introduction

The aim of Assignment 1 is to build an efficient Boolean retrieval system for English corpora using efficient inverted index structures.

1.1 Input

Given is a document collection-extracted from a benchmark collection in TREC.
Document - Identified by an alphanumeric Document ID, represented as an XML Fragment.
XML Tags - Contains the indexable portion of the document.

1.2 Task

The task to build an efficient Boolean Retrieval system can be broken down into three subtasks:
Inverted Index Creation: Inverted the document collection and built an on-disk inverted index. The on-disk inverted index consists of a dictionary file and a single file for all postings lists.
Compression: To reduce space used to store postings lists, used compression schemes c1,c2 and c3.
Boolean Retrieval: Using inverted index structures implemented single and multi keyword retrieval.

2 Program Structure

The aim of the assignment is achieved in a two step process. At first the inverted index data structure is created followed by Boolean search. Below is described the implementation of the two processes separately.

2.1 Inverted indexing of the collection

Program is named as invidx_cons.py. The implementation is described below.

1. **Pre-Processing Dataset** : The function *preprocess(all_files,required_tags,docidentifier,stopwords,filepath)* takes the directory path, xml-tags containing the indexable portion, tag used as a document identifier, list of stopwords and the name of the directory as arguments to parse the entire document collection and generates a token to docIDs dictionary.

   Since the docIDs are alphanumeric, an integer is assigned to each docID which is used as an aliter to the original docIDs. This mapping is stored in a dictionary which is also generated by this function.

   In this function, each document is parsed using the BeautifulSoup library. All relevant tags containing the indexable portion are used to extract the indexable content.The indexable content is tokenized by splitting across punctuations (',.:;()'"), spaces and new line character. The generated list of tokenized content is passed in a stemming algorithm (PorterStemmer) reducing the variant forms of a word to a common form. A dictionary is maintained with the key as a word and the value as a list of the docID's (denoted by the assigned integer docID) containing the specific word. For each document ID a Boolean dictionary is also maintained. If a word is present in the document, the value of the word in Boolean dictionary is set to 1. Creating this dictionary eases the process of eliminating occurrence of multiple words in the same document.

2. **Creating Posting Lists** : This step involves the creation of a common postings list and a dictionary. An empty list, say post.list is instantiated. For each word in the above created dictionary, the length of it's postings list is calculated, and the postings list is added to the common list, post.list. The dictionary has a structure with the word in the vocabulary as the key and a list of length 2 as the value. Suppose a term in vocabulary is "temp", then dict[temp] = [a1,a2]. a1 denotes the starting position of the postings list of temp in post.list, and a2 defines the length of the postings list. a2 serves as an offset for the next term to be added in the dictionary.

3. **Compression** : The first step of any compression scheme is to apply Gap Encoding to the original list. Gap Encoding : Gaps between docIDs are much shorter than the docIDs and for frequent words usually equals 1, which requires much less space than original docIDs. The common postings list (post.list) is reduced to a gap encoded list.

   C0 compression : Since according to the assignment constraints, it could be assumed that the docIDs would fit in a 32 bit unsigned integer, 32 bits are allocated to the encoding, which is the binary representation, of each integer in the postings list. For decoding the list is processed in skip sizes of 32, converting each chunk into it's integer. The dictionary dict is changed as per the new start and offset positions as well, to aid during decoding.

   C1 compression : For this compression scheme, a mask (1111111) is created. The BITWISE AND of the mask and an integer, say k, generates the 7 LSB bits of k. The 7 LSB bits are converted into it's binary form with a fixed length of 7. The right shift operator $\gg$ is applied to k to process the next chunk. Continuation bits are added alongside as well. For decoding, the binary representation is traversed in skip sizes of 8, converting each chunk into it's integer until a 0 at MSB is found.

   C2 compression : Each integer is encoded with O(log x) bits using the encoding expression :

   U(l(l(x)) + lsb(l(x),l(l(x))-1) + lsb(x,l(x)-1)

   where + denotes concatenation of bits, U denotes the unary representation, l(x) denotes length of bin(x) while lsb(a,b) denotes the b least significant digits of the binary representation of a.

   For decoding, we start with identifying the unary code which gives us l(l(x)) as per the encoding expression, reading further l(l(x))-1 bits gives the lsb(l(x),l(l(x))-1) which gives l(x) (without 1 at MSB) and the next l(x)-1 digits provides the binary representation of x (without 1 at MSB).

   C3 compression : For C3, snappy library is used. The encoding of C0 (no compression) is passed to snappy, which generates encoded bytes for the posting list. For decoding, the snappy library is used to uncompress the compressed byte file.

4. **Writing to File :** If the generated encoded string is written directly to a file, each entry ("0" or "1") will occupy a single byte thereby consuming a lot of space. To resolve this, every chunk of 8 bits is converted into an integer, this integer is converted into a byte instance which is written to the file. The dictionary contains the information of the docID to integer mapping, the compression used and other additional information used for decoding in a nested dictionary structure. For writing C2, the length of the encoded postings list is converted first into a multiple of 8 by adding required "0"s to the end of the list. The number of "0"s added is stored to the dictionary as well.

2.2 Boolean Search Retrieval

Program is named boolsearch.sh. The implementation is described below.

1. **Extracting Compressed Postings List** : From the file indexfile.dict the type of compression used is extracted. If the compression used is not C3, the file is read byte by byte to generate the decompressed postings list. The number of added "0"s are skipped, to finally generate the postings list.

2. **Pre-Processing Query** : Each query is first pre-processed to remove any stopwords and the PorterStemmer algorithm is used to stem all the words of the query.

3. **Keyword Retrieval** : For searching the documents containing all the words of the query. The document to integer mapping is retrieved from the dict to convert the integers back to their corresponding docIDs.

Single Keyword Retrieval : Searching for a single keyword is equivalent to decoding it's postings list. Using the stored dictionary, the start and the length of the postings list of the term in the compressed file is taken. The part from the start to start+length is decoded to retrieve the posting list of the term.

Multi Keyword Retrieval : For multi retrieval, a linear merge algorithm is applied to calculate the intersection of all the individual posting lists of the words which runs in O(l1+l2), where l1, l2 are the lists to be merged.

3 Metrics Of Interest

3.1 Index Size Ratio (ISR)

ISR is computed as the ratio of the sum of the dictionary, the postings list file and the collection size, all expressed in bytes. Collection size = 51,63,21,280 bytes.

1. **C0 Compression** : Size of Dictionary = 1,35,04,512 , Size of Postings List = 14,70,91,456 .
   ISR = 0.3110
2. **C1 Compression** : Size of Dictionary = 1,31,76,832 , Size of Postings List = 4,17,83,296 .
   ISR = 0.1065
3. **C2 Compression** : Size of Dictionary = 1,31,76,832 , Size of Postings List = 3,37,83,808 .
   ISR = 0.0901
4. **C3 Compression** : Size of Dictionary = 1,25,48,414 , Size of Postings List = 5,27,40,096 .
   ISR = 0.1265

3.2 Compression Speed

1. **C0 Compression** : C0 compression time - 78.59 * 10^3 millisecs
2. **C1 Compression** : C1 compression time - 95.39 * 10^3 millisecs
   Compression speed : C1 - C0 = 95.39 - 78.59 = 16.8 * 10^3 millisecs
3. **C2 Compression** : C2 compression time - 130.58 * 10^3 millisecs
   Compression speed : C2 - C0 = 130.58 - 78.59 = 51.99 * 10^3 millisecs
4. **C3 Compression** : C3 compression time - 19.35 * 10^3 millisecs
   Compression speed : C3 - C0 = 19.349 - 78.59 = -59.241 * 10^3 millisecs

3.3 Query Speed

1. **C0 Compression** : Number of Queries = 50 , Total Query Time = 236.56 secs .
   Average Query Time = 4.73 * 10^6 μs
2. **C1 Compression** : Number of Queries = 50 , Total Query Time = 52.167 .
   Average Query Time = 1.04334 * 10^6 μs
3. **C2 Compression** : Number of Queries = 50 , Total Query Time = 39.157 secs .
   Average Query Time = 0.78314 * 10^6 μs
4. **C3 Compression** : Number of Queries = 50 , Total Query Time = 11.39 secs .
   Average Query Time = 0.2278 * 10^6 μs