INFORMATION RETRIEVAL PROJECT

BOOLEAN SEARCH WITH SINGLE PASS IN-MEMORY INDEXING

August 23, 2018

Student ID: 20174797
Student Name: Baturay Ofluoglu
University of Antwerp
Computer Science: Data Science

Contents

1 Preprocessing of Document Collection **4**
1.1 Punctuation Removal 4
1.2 Number Removal 4
1.3 Lowercase Conversion 4
1.4 Stop-word Removal 4
1.5 Stemming 5
1.6 Preprocessing Implementation 5

2 Single Pass in Memory Indexing (SPIMI) **6**
2.1 Inverting Blocks 6
2.1.1 Implementation of Inverting Blocks 7
2.2 Sorting Blocks 7
2.2.1 Implementation of Sorting Blocks 7
2.3 Merging Blocks 8
2.3.1 Implementation of Merging Blocks 9

3 COMPRESSING SPIMI **11**
3.1 Variable Byte Encoding 11
3.1.1 Implementation of Variable Byte Encoding 12
3.2 Gamma Encoding 12
3.2.1 Gamma Encoding Implementation 13

4 BOOLEAN RETRIEVAL **14**
4.1 Implementation of Boolean Retrieval 14

5 Summary of Boolean Retrieval with SPIMI Algorithm **15**

6 Conclusion **15**

INTRODUCTION

The purpose of this project is to implement Single Pass in Memory Indexing (SPIMI) for efficient boolean search. SPIMI is a scalable indexing technique that enables to index any data that fits into the disc. There are also compression extensions to SPIMI algorithm to decrease indexing time and boolean searching time. In this report, we first explain SPIMI algorithm and the compression extensions. We then move to how SPIMI algorithm can be used to answer boolean queries. Finally, we will conclude by mentioning about the performance of SPIMI algorithm.

THE GENERAL OVERVIEW OF THE PROJECT

1. Preprocessing of Document Collection
2. Single Pass in Memory Indexing
3. Compressing SPIMI
4. Boolean Retrieval
5. Summary of Boolean Retrieval with SPIMI Algorithm
6. Conclusion

1 Preprocessing of Document Collection

Preprocessing of the keys decrease the total number of terms in the collection significantly. We use five different preprocessing techniques which are listed below. In our collection, there are 4368 email documents. The collection has 92886 terms in total, and we apply the following preprocessing techniques to decrease the total number of terms to store less indexing data.

1.1 Punctuation Removal

Punctuation removal is a significant step to tokenize documents correctly. When we tokenize the following phrases based on white space split, there will be ’Turing’ and ’Turing.’ keys. However, both keys are the same. We have many similar examples as this case, and therefore the number of terms decreased significantly from 92886 to 52784 after applying punctuation removal method.

- Alan Turing is the father of AI. The first computer is found by Alan Turing.

1.2 Number Removal

There are an infinite amount of numbers. For this reason, the frequency of a specific number in the collection is very low. Also, storing all distinct numbers in our index file as a key is an expensive operation. Therefore, we prefer to remove all the numbers from the documents for performance concern. The number removal method decreases the total amount of terms from 52,784 to 43,908.

1.3 Lowercase Conversion

The terms at the beginning of each sentence start with capital letters. For example, ‘Information’ and ‘information’ words are the same, but the algorithm saves those words as a different key. Therefore, we convert all uppercase letters to lowercase. This operation eliminates 8,368 terms. So, the excluded terms drop to 35,540.

1.4 Stop-word Removal

In English, there are approximately 150 stop words. For Boolean search, these stop words are not necessary to increase the quality of the search. Thus, we eliminate these words. This step filters out 150 terms.

1.5 Stemming

There may be some extensions to the root of the word in most of the languages as a part of a grammar rule. For example, the end of the English words may end with '-ing' or '-s' based on English grammar rules. Stemming target to remove these kinds of grammar extensions. By applying the stemming technique for example 'coding' and 'code' will be indexed under the same word.

We intentionally use this technique at the end because it is the most expensive preprocessing operation. After eliminating most of the words, doing stemming takes much less execution time. After stemming activity, the total number of words decreased to 25,379.

1.6 Preprocessing Implementation

The document collection needs to be at the `Emails\Raw` path. Also, all the documents must be in '.txt' format. The users do not have to define any document id.

The algorithm first finds all the 'txt' files from `Emails\Raw` path and assigns each file a unique file id. The id is stored at the file name. If the file name is 'abc.txt', it will be converted to '1.txt' and '1' symbolize the document id. Note that the document ids start from 1 because it is hard to identify 0 in a variable byte and gamma encoding.

After changing all file names, the preprocessing step begins for all *txt* files before running SPIMI algorithm. The preprocessed files will be recorded to `Emails\Preprocessed` path. Because we will do a boolean search, we do not need duplicate words at the documents to check the word frequency. For this reason, we remove the duplicates, and we record the set of terms for each document. This process saves a significant amount of disc space, and it will decrease the following SPIMI indexing step. Preprocessing the document collection takes approximately 6.9 minutes.

Note that after completing preprocessing, we write a dummy flag file called *000_flag.flg* to `Emails\Raw` path. If the code recognizes this file in that file, it means that preprocessing has been done before and it does not preprocess the raw documents again to save time. To preprocess the documents again, delete the flag file and rerun the code.

The implementation steps of the preprocessing are straightforward. It can be checked from *preprocess_docs.py* file.

2 Single Pass in Memory Indexing (SPIMI)

SPIMI algorithm is a widely used indexing technique in Information Retrieval. Its main advantage over other indexing techniques such as BSBI is its scalability property. If the documents cannot be imported into main memory entirely due to the capacity, then SPIMI algorithm can handle memory constraint. In the following parts, we will explain how SPIMI can overcome memory.

The algorithm has three main steps which are inverting, sorting and merging blocks.

2.1 Inverting Blocks

In this step, the algorithm starts reading and tokenizing documents from the collection. It saves tokens as a dictionary key and document IDs as dictionary values. Dictionary values are also called 'posting lists'. The memory accumulates the keys and posting lists until the memory gets full. Let's invert the following five documents with SPIMI algorithm.

- d1: can a machine think
- d2: machine learning
- d3: game theory
- d4: deep learning
- d5: imitation game

Suppose we have limited memory and two blocks are sufficient for inverting. The first block can store three documents and the second block can store the rest. For this example, assume that we do not use preprocessing. In this case, the inverted blocks will look at table 1.

Table 1: Inverted Blocks

1st Block	**2nd Block**
can: 1	deep: 4
a: 1	learning: 4
machine: 1, 2	imitation: 5
think:1	game: 5
learning: 2	
game: 3	
theory: 3	

2.1.1 Implementation of Inverting Blocks

Before inversion, we define a memory limit regarding bytes. In this case, we assume that our memory has *100,000* bytes capacity thus the blocks cannot exceed this amount. If it exceeds the limit, we skip to the next sorting stage.

While inserting keys and values, we use *defaultdict* object in python. We search for the keys in the dictionary. If it exists, we append the document ID into keys' value (posting list). Otherwise, we create a new key and add document id into its value. These operations have $O(1)$ complexity because searching is done via hashing in defaultdict and appending a value to a list also takes $O(1)$ time.

We always check the defaultdict size in main memory. If it exceeds our limit, we skip to the next sorting stage.

2.2 Sorting Blocks

After inverting a block, we sort the keys lexicographically. Then, we write the sorted block into the disc and flush the memory to be able to create the next block. Sorting the blocks will provide us 'a single pass' to the disc that we will give more details at the next merging blocks section. Refer to the next table for the sorted version of inverted blocks.

Table 2: Sorted Blocks

1st Block	**2nd Block**
a: 1	
can: 1	deep: 4
game: 3	game: 5
machine: 1,2	imitation: 5
learning: 2	learning: 4
theory: 3	
think:1	

2.2.1 Implementation of Sorting Blocks

Sorting blocks implementation is straightforward. We first sort *defaultdict* object keys in lexicographic order and save them in a sorted list of lists. Each list element includes key and postings in a list.

After sorting, we write the list elements line by line into a text file in the following format. The inverted text files are stored in `\Blocks\` folder.

- *Key1:{Doc ID1, Doc ID2}*
- *Key2:{Doc ID2, Doc ID3}*

We use curly brackets to identify the posting lists. The curly brackets are not beneficial for the uncompressed posting lists, but we need them to decompress posting lists to obtain postings successfully.

Each written block has an integer ID starting from 1 at the file names. For example, the first created inverted index file name is *1.txt*. The block ID is essential in merging step to merge the index files in order.

2.3 Merging Blocks

After reading all documents and saving sorted blocks into the disc, we need to merge all blocks to create the final index file. The created blocks are merged two by two with logarithmic merging technique because it is much efficient than linear merging. Linear merging has $O(n)$ complexity while logarithmic merging has $O(\log n)$ complexity. Refer to figure 2 for logarithmic merging example. Also, note that the merging order is important. For instance, firstly created block can only be merged with the secondly created block. Otherwise, we cannot retrieve the posting lists correctly if we use compression.

While merging we read the blocks line by line and after each line reading, we make merging operation if necessary. So, we do not exceed the memory in this way.

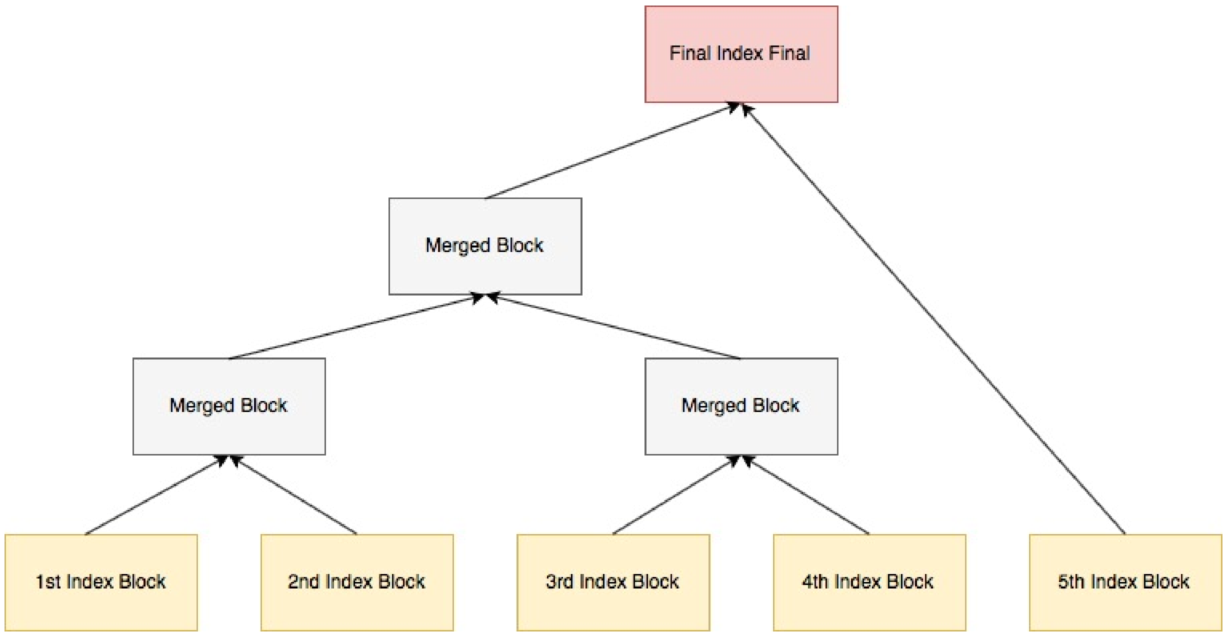
The details of merging blocks are given in the next implementation section. If we merge first and second blocks at table 2, the merged block will look like table 3.

Table 3: Merged Blocks

1st Block	**2nd Block**	**Merged Block**
a: 1	deep: 4	a: 1
can: 1	game: 5	can: 1
game: 3	imitation: 5	deep: 4
machine: 1,2	learning: 4	game: 3,5
learning: 2		imitation: 5
theory: 3		machine: 1, 2
think:1		learning: 2, 4
		theory: 3
		think: 1

.

Figure 1: Merging Inverted&Sorted Index Files



2.3.1 Implementation of Merging Blocks

Note that each block can barely fit in memory. So, we cannot load two blocks in memory. Instead, we need to read each key and posting pairs one by one from two blocks. The algorithm to merge all inverted index files is written below.

Notations:

M: The text file that contains merged two index files

B: Buffer

P1: Posting list of 1st block

K1: Key of 1st block

P2: Posting list of 2nd block

K2: Key of 2nd block

Algorithm to Merge Two Blocks:

1. Create empty *M* file
2. Read next key and posting pair from the first and second block.
3. if K1 < K2 lexicographically, add K1 and P1 into B
4. else if K1 > K2 lexicographically, add K2 and P2 into B
5. else if K1 = K2, add K1 and P1.append(P2) into B. Note that the first document id of the posting lists at the second block is always greater than the last document id of the first blocks' posting list. So, append P2 into P1.
6. else if K1 is None (first block is completely read) and K2 is not None, add K2 and P2 into B
7. else if K2 is None (the second block is completely read) and K1 is not None, add K2 and P2 into B
8. else if K2 and K1 is None (Two blocks are read)
   (a) delete first and second block from disc
   (b) append all key and posting list pairs at B into M
   (c) change the file name of M as *_[first block name].txt* and break the code
9. if B is greater than memory limit, write B into M in 'a' (appending) mode and flush the memory space. Go back to step 2
10. else go back to step 2

Algorithm to Merge All Blocks:

1. List all txt file names from the `Blocks/` path
2. Extract document ids from the filenames and sort the filenames list based on ID values in increasing order.
3. If *mod(length of sorted list, 2) == 0*, create pairs list $(i, i+1) \forall i$ where $0 \leq i <$ *length of sorted list* - 1, $i \in \mathbb{Z}$
4. Else if *mod(length of sorted list, 2) == 1*, create pairs list $(i, i+1) \forall i$ where $0 \leq i <$ *length of sorted list - 2*, $i \in \mathbb{Z}$

5. Merge all block pairs by using merging two blocks algorithm written above (E.g the first pair will be seen as (0,1), it indicates the position of blocks in the sorted list. So, from sorted list take the first and second block names and merge them.)
6. If `Blocks/` path has not one *.txt* file, go back to step 1
7. Else change the file name either *vb_index.txt* or *gamma_index.txt* based on compression type and return the final merged index file

3 COMPRESSING SPIMI

SPIMI indexing occupies a significant amount of disc space. With compression techniques, it is possible to compress the data by 75%. Compression has several advantages in addition to disc space saving. It decreases the indexing time because while indexing more data can be stored in memory thus the OS do less I/O operations. Because more information can be stored in main memory, it will also decrease the searching time. The main idea is that loading compressed data into memory and decompressing it takes less time than loading uncompressed data into memory.

In compression, we can compress both keys and posting lists. Compressing posting lists save much more space than compressing the keys. In this project, we compress the posting lists, but we did not compress the keys. However, we did preprocessing to decrease the total number of keys. For the posting list compression, we use variable byte encoding and gamma encoding to find the more efficient algorithm for posting list compression.

Before applying variable byte or gamma encoding, instead of storing all document ids we store the gaps between the document ids. Thanks to gap encoding, the compression algorithms compress much fewer data, and it decreases the compression execution time. Also, the posting lists occupy much less data on the disc. Let's give an example to the gap encoding. Suppose the posting list is [10000, 10005, 10010, 10019]. If we apply gap encoding, the new posting list will be [10000, 5, 5, 9]. Gap encoding increases the compression algorithms performance significantly.

3.1 Variable Byte Encoding

Variable byte encoding is used to encode posting list gaps. Without using any compression algorithms, we need to use 4 bytes or 32-bit integers to encode gaps. However, most of the time we do not need to use 4 bytes. Variable Byte Encoding provides a flexible amount of bytes based on the gap amount. For example, 2^2 needs only 1 byte while 2^{10} needs 2 bytes to encode.

The problem is that we cannot tell the computer that it must use the flexible number of bytes for each gap value. We can instead record a byte stream as a kind of string. When we need, we can decode that byte stream to obtain the real gap values. Another problem is how we can split the gaps from the byte stream. Variable byte encoding handles all these problems and offers dynamic byte encoding.

In variable byte encoding, the last 7 bits are used to encode the numbers and the first bit is the continuation bit which is used for splitting the gaps. Let's analyze the example below.

- 824: 0000011010111000

824 is encoded with two bytes. As it can be seen, the first bit is 0. It means that we have a new gap value. In other words, it is the splitting condition. In the second byte, we see that the first bit is 1. It means that do not split and continue encoding. The other 7 bits are used to encode '824' value. So instead of 4 bytes encoding, we succeed to encode 824 with only 2 bytes. Let's look at another posting example. Suppose our posting list is [824,4,3]. Then it will be encoded as follows.

- [824,4,3] = 00000110 10111000 00000100 00000011

We only used 4 bytes for three integers encoding in this case thanks to the byte stream.

3.1.1 Implementation of Variable Byte Encoding

Variable byte encoding and decoding implementation can be found in `Compress/VariableByteEncode.py` file. Note that to be able to index the collection with variable byte encoding, you need to run compressed_spimi_vbencode.py file.

3.2 Gamma Encoding

In gamma encoding, the purpose is to use dynamic bits for each gap instead of fix number of bytes. Differently, from variable byte encoding, we do not have byte streams we have bitstreams. In variable byte encoding, even we do not need, the total number of bits should be multiples of 8. For example, '2' is encoded as '00000010' in variable byte encoding but the first 6 bits are unnecessary. Gamma encoding, however, uses bit streams and it does not have to append unnecessary 0's to create a byte.

If gamma encoding uses bitstreams then the question is how it splits the gaps. Just as variable byte encoding, gamma encoding has both continuation bits and gap encoding part. This pair is called as length and offset. For the continuation bits, it uses unary encoding. In unary encode, we need to add N ones and one zero to the end where N is an integer. The N symbolize the length value.

Let's give an example to understand better. Suppose we have a posting list which is `[2, 3]`. Without using a splitting condition, the encoding would be 1011. So, it is impossible to decode this value because we do not know from where we should split the bit stream. For somehow, we need to keep the length of the gap numbers such as the first 2 bits is a number and the last 2 bits symbolize another gap number. We should also keep the value of the gap. We use binary encoding to keep the values of the gaps. However, we drop the leading 1 from each gap value. This part is called as the offset. In the end, we append length and offset so 2 will be encoded as 100 and 3 will be 101. In a bit stream, we can show the gamma encoded posting list as 100101. Now, we can split the gaps for decoding. The promising part is that in variable byte coding we encode 2 and 3 with 16 bits in total but in this case, we only use 6 bits.

3.2.1 Gamma Encoding Implementation

Gamma encoding and decoding implementation can be found in `Compress/GammaCode.py` file. Note that to be able to index the collection with gamma encoding, run *compressed_spimi_gamma.py file*.

In theory, gamma encoding is stored in , but we cannot write bit streams into a text file. We can write bytes in python. So, we convert bit stream to byte stream. Converting bit stream requires to append or prepend 0s to the bit stream to make the length multiples of 8. We prefer prepending in our case. The problem is we need to know how many 0s are prepended during the decoding phase. Otherwise, they will be decoded as 1 which is not true.

We prepend N zeros to the beginning of the bit stream to make it a multiple of 8 and convert it to a byte stream from bitstream. Also, we convert N to one byte because the largest value of N can be 7. After, we append that byte to the end of the byte stream. So, each posting lists will have two more bytes at the worst case.

While decoding, we first convert the last byte to an integer. Thanks to last bytes' integer value, we understand the total number of prepended 0s, and we will be able to obtain real bitstream for the posting list.

4 BOOLEAN RETRIEVAL

The general idea in boolean retrieval is to split the query into terms and search each term from the index files' key and return its posting lists.

In big index files, the problem is that index file cannot be loaded into main memory due to the capacity constraint. Therefore, we cannot do an efficient search in case we try to load the entire index file. So, we split the index file into small pieces (20 KB per piece). For a given query term, we first find the relevant split index file, and then we start to iterate over all keys until we find the query term. When we find, we return the posting list of that term to the user. Note that if the query has multiple terms, then we find the intersection of all retrieved posting lists and return the intersection to the user. More information is provided at the following implementation section.

4.1 Implementation of Boolean Retrieval

We prefer to split big index files into 20 KB of pages or splits. The splits are stored in `Index/` path

Finding the relevant index piece for a query term is problematic. In our approach, we assign the last key to the split filename to solve this problem. For example, if the last key of a file is 'computer' then the split file name will be 'computer' without any file extension. After naming all split files, we sort them in lexicographic order. We also change the last filename as *'zzzzzzzzzzzz'* to indicate the last file. Suppose the split file names are [*computer, information, science, zzzzzzzzzzzz*]. In this case, for a given query *'linux'*, ***science*** file will be selected because *information* < ***linux*** < *science.* So, if we apply linear searching to find the relevant index file, the complexity would be $O(n)$. We prefer to use b+ trees to find the relevant index file. The B+ tree is a balanced tree data structure, and it is generally used for range queries. Our problem is also a type of range query. Thanks to this balanced tree, we can find the relevant index file with $O(\log n)$ complexity.

After finding the relevant index file efficiently, we will read the key and posting pairs one by one until we find a match with the query term. If we find a match, then we will decode and return the posting list.

If we have a query with multiple terms, then we find posting list for each term, and we need to find common document ids from all posting lists. To find the intersection of the posting lists, we add posting lists into an empty list, and we sort them based on the amount of document id in each posting. The posting lists that has the least amount of document ID will appear at first. Then, starting from the first element, we find the set intersections. Starting intersection from the posting list that has least document id decreases the execution time because the algorithm will go over fewer document IDs.

5 Summary of Boolean Retrieval with SPIMI Algorithm

1. Get input from the user for compression type [VBEncode/Gamma]
2. Pre-process all files at the `Emails/Raw/` and save preprocess files to `Emails/Preprocessed/` path
3. Invert and sort all files with the compression type specified at step 1
4. Merge all inverted index files with logarithmic merging technique
5. Split merged index file into 20KB of chunks
6. Build B+ tree for index chunks' file names
7. Get query input from the user
8. Pre-process query input
9. For each query term find relevant index file by using B+ tree
10. Find the posting list from the found index file and decode it
11. Find intersection of all retrieved posting lists for all preprocessed query terms
12. Return intersected document ids to the user, go back to step 7 to search a new query

6 Conclusion

In this project, our primary purpose was to build an index file to do the boolean search as fast as we can. Therefore, we first preprocess the collection to decrease the number of terms. We succeed to decrease the terms from 92,886 to 25,379. Also, we used posting list compression techniques to decrease the indexing and searching time because of memory constraints.

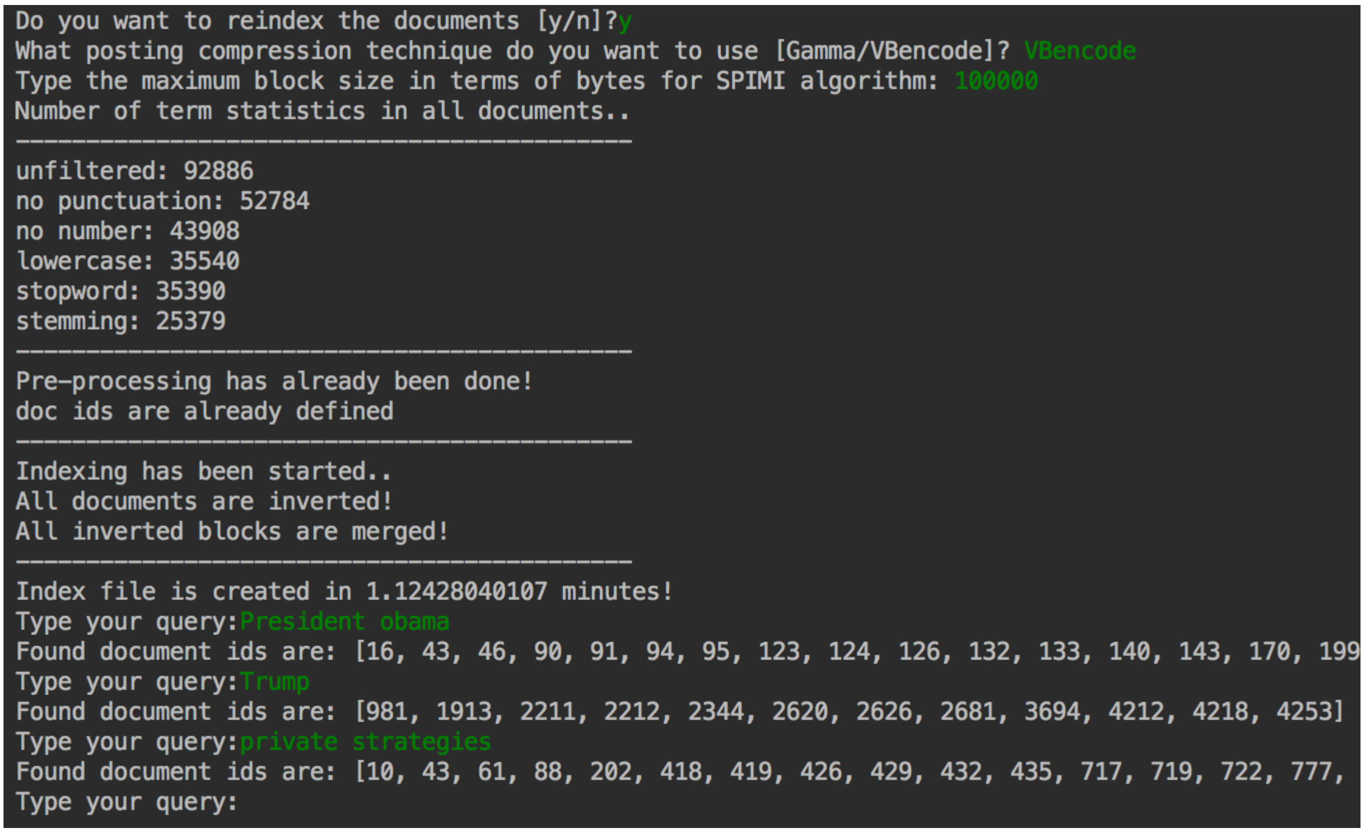
We tried two different compression techniques which are variable byte encoding and gamma encoding.

Our SPIMI implementation with variable byte encoding index 92,886 documents in 1.12 minutes if we allocate 100,000-byte memory space. The total capacity of the Spimi index occupies 876,113 bytes in the disc. On the other hand, the Spimi implementation with gamma encoding execution time is 7.14 minutes. However, compared to Variable byte encoding it occupies approximately 10% less space on disc with 790,460. So, the boolean search would be faster in gamma encoding.

Both variable byte and gamma encoding have its advantages. If the collection is massive and indexing time is crucial then using variable byte is more suitable. If the indexing time is not that important and query search needs to be faster, then using gamma encoding leads to better results. Note that with key compression techniques, the index file size can be even more compressed but it is not implemented in this project.

APPENDIX

Figure 2: Screen Shot From The Console Program



```
Do you want to reindex the documents [y/n]?y
What posting compression technique do you want to use [Gamma/VBencode]? VBencode
Type the maximum block size in terms of bytes for SPIMI algorithm: 100000
Number of term statistics in all documents..
--------------------------------------------
unfiltered: 92886
no punctuation: 52784
no number: 43908
lowercase: 35540
stopword: 35390
stemming: 25379
--------------------------------------------
Pre-processing has already been done!
doc ids are already defined
--------------------------------------------
Indexing has been started..
All documents are inverted!
All inverted blocks are merged!
--------------------------------------------
Index file is created in 1.12428040107 minutes!
Type your query:President obama
Found document ids are: [16, 43, 46, 90, 91, 94, 95, 123, 124, 126, 132, 133, 140, 143, 170, 199
Type your query:Trump
Found document ids are: [981, 1913, 2211, 2212, 2344, 2620, 2626, 2681, 3694, 4212, 4218, 4253]
Type your query:private strategies
Found document ids are: [10, 43, 61, 88, 202, 418, 419, 426, 429, 432, 435, 717, 719, 722, 777,
Type your query:
```