8/4/2019

An examination of inverted lists and search querying

An implementation

Ryan Chew (s3714984), Jin Zeng (s3688213)
RMIT UNIVERSITY

Table of Contents

1. Introduction

The action of searching through thousands of documents for a single word in a term is a simple concept, one so simple, that it should not be thought of to be very complicated. Furthermore, this should all be done no longer than a fraction of a second. A huge amount of research and development has gone into creating faster ways to complete this task. Here, the focus is on developing a basic inverted index and querying program.

An *inverted index*[1] is a data structure of documents, consisting of words, which documents and the frequency of the words in the documents. Though many implementations of inverted indexes may also contain information on the location of the words in each document, it was not implemented in this version.

This paper discusses the decisions made in the implementation of the inverted index and related modules, and the querying module.

The rest of the paper is organized as follows. The next section discusses index construction and the decisions made regarding handling and processing the text data. In Section 3, the stoplist module will be described, as well as implementation thoughts. Section 4 will focus on the query mechanism and how the function works, and the direction taken. Section 5 explores the size difference between the produced files and the original documents.

2. Index Construction

This section will contain explain methods and decisions pertaining to document parsing methods. How the data is merged together for further processing to the construction of the inverted index. The formatting of the data written to disk.

2.1 Tokenization of terms

The tokenization and normalization of terms is quite important in the fast and efficient indexing, querying of data.

The method used to process the data is very simple. During the progression of reading the lines of the given collection of documents, the normalization of the text was performed in parallel.

Firstly, general punctuation removal was performed using java Regular Expressions (regex). A simple statement[2] checks for any punctuation that contains non-white-space characters on either side, as separate checks. This allows the program to easily clear commas, quotation-marks, etc.

Decisions regarding text normalization such as dealing with hyphenated words, and acronyms were done with word isolation in mind. It was decided that hyphenated words would be split to their separate words for simplicity. Acronyms would be joined together to remove the need for punctuation processing. The process of removing hyphens and dots is like the above regex methods. Using a basic expression for capturing a hyphen and full-stop character, the characters were replaced with spaces for hyphens, and were outright removed for full-stops.

Continuing using the available methods in the java standard library, a set regex term was constructed to efficiently parse out punctuation and standard word contractions (aren't -> are not). This set of regex terms were applied to the inputted header and body text for processing prior to internal storage.

Other actions such as *case-folding* were performed in tandem with the regex utilizing existing Java *toLowerCase* functionality.

After the text has been processed, the program will create an internal document for further processing.

[1] (Black, 2017)

[2] Source code: DocumentHandler, ln 167

2.2 Statistical frequency gathering and data merging

A key step in the creation of inverted index lists is the gathering of the frequency of terms in the document collection.

The implementation of the gathering mechanism in the program employs a HashMap, used for its more efficient non-synchronized nature.

The program iterates through the collection of documents, extracting the heading and text content data, splitting them into a list of strings before running them through the checker program.

The checker module iterates through the list of given text data and checks if the hash-map contains a term. If that term does not exist it will create a new entry for the term in the hash-map, or if that entry does exist but not the document in that terms collection of documents, it will create a new entry for the document under that term. Otherwise, it will increment the current counter for that document under that term. This implementation of the hash-map allows for quick storage and retrieval of the words and the document data. After the collection of terms from the documents, the empty string term is removed.

2.3 Final file format of outputted files

The output files written are in the format of standard data files (no extension).

3. Stoplist

The stoplist input was taken from the inputted stop list file. In parallel to reading the file, several hashing functions were tested, using the DigestMessage module, for performance. However, for the end product, the default hashing function was chosen.

3.1 Currently used hash function

The currently used hash in this program is the standard Java hashcode function. This function has been optimized very well for the purposes proposed for this program, string hashing. The implementation for Javas hashcode function prioritizes avoiding collisions for the dataset. The function utilizes the prime number, 31, in its calculation of the hashcode. Iterating through each letter in the given string. Furthermore, Javas hashmap hashing implementation trades performance with safety in terms of bit spreading and collision avoidance. By XORing the upper bits of the hashcode with the lower bits, the hashmap hash function spreads the impact of higher bits to the lower bits to avoid collisions of data that vary only in the higher bits that are outside the current power-of-two mask.

This is a good enough hash for the purpose of the program.

3.2 Other hash functions

Other hashing functions were tested for their performance. It should be noted that these tests are double hashed, that is to say after these hashes are performed, a further hashcode function is performed on top due to the implementation of Javas hashmap. The tests were run on the same data set. The results are as follows;

Algorithm	Time trials (ms)			
	1	2	3	Average
Java HashCode	454	485	423	457
MD5	1600	1499	1486	1528
SHA1	1665	1753	1845	1754
SHA256	2100	2005	2031	2045

Table 1 - Timings of hashing algorithms compared, timed with the complete program running. Ran on Windows 10, Ryzen 5 2600 processor

In Table 1, the average timings of the hashing algorithms are compared. The timings of the more complex algorithms (MD5, SHA1, SHA256) take much longer than the in-built java hash code. Comparing the overall time differences, the MD5 algorithm was decidedly the more efficient function. The reasoning behind this decision lies within the fact that there is no need for a higher bit

hash, and there is also no need for security in this data. SHA1 and SHA256 processing requirements increase highly over larger datasets and are not needed. Hence, the decision to use MD5.

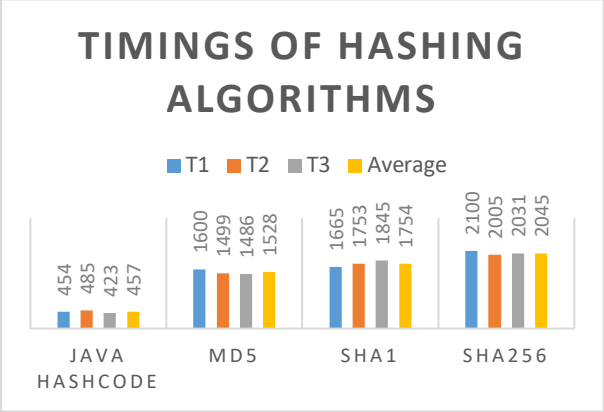

Figure 1 - Graphic of Table 1 data

From the above graphic, it can be derived that with more complex hashing algorithms the overall increase in time moves quite linear. The trade from complexity against time further reinforces the decision to use MD5 over the other algorithms.

MD5 is a standard, widely used hash for verifying data that generates a 128-bit hash value. This hash fulfils the needs of the program enough that it was not needed to use higher bit hashing functions.

3.3 Implementation

The way the stopping functionality was implemented included, reading the stoplist into file, then just storing the words into a hashmap. Then using this hashmap as a reference when going through the text data. To remove the stop words. This was a simple implementation that took advantage of the O(1) runtime of hashmap structures for high efficiency.

4. Search Querying

The data structure that was used to hold the lexicon for processing query terms was carefully thought out to allow easy access and fast processing. Two hash maps were used to efficiently store keys and values. The first hash map was created to map the query term with the number of documents associated with the term, and the byte offset, to the document frequency information, in the inverted list.

An auxiliary class was created as a workaround for the limitations of the hash map class, to facilitate the storage of the related attributes of the term, so that 3 values could be held by each entry in the hashmap. The mapping table hash map was created to map the ordered document Id with the raw document names, to generate a more efficient search result. Although in real world scenarios, the value would be the byte offset to the document in the original collection. After the hash map creation, the next step would be to extract the document ID and term frequency based upon the byte offset value from the Lexicon hash map. In order to only read the appropriate chunk from the file, a built-in function, 'seek' in the *RandomAccessFile* class was used to point at a specific byte in the inverted list to avoid sequentially scanning from the start of the file.

The documentID and term frequency, were read with a built-in function 'read'. After getting the document number from the inverted list, it was then used to get the raw document name from the mapping table.

5. Index Size

The sizes of the outputted data were recorded for comparison on different systems.

System	Invlist	Lexicon	map
MacOSX	157KB	154KB	2.0KB
Windows	157KB	154KB	2.0KB
Linux	157KB	154KB	1.7KB

Table 2 - Sizes of outputted files on different major OSs

The above table describes the files sizes of the output files. The output files are much smaller than the original input file (~400KB). This is largely due to the amount of removed data from the original file.

With compression, the amount of space saved would be even greater. The program would take longer to perform, but the

required storage of the output files would be less.

6. Compression

Inverted indexes sizes grow in size quite fast and certainly need to undergo some form of compression to be used in practical situations.

The chosen compression algorithm for this program is **Variable Byte Encoding**. The implementation in this program utilizes bitwise operators to handle binary calculations to get the individual bytes for processing. The program iterates through the inputted number 7 bits at a time

The difference in size is quite substantial where the original size of the inverted list was around 157KB, after compression the file became 39KB. This is a ~4 compression ratio with a ~75% saving rate.

3.1 Methodology

The architecture of the program created made use of a simple strategy pattern design that allowed for a streamlined process in the compression and decompression. Having a generic 'strategy' and an abstract implementation of the strategy for shared functions, this design was able to be used to allow for generic calls and a simpler design overall. Utilizing bit shift mechanics the program was able to repeatedly easily build byte chunks quickly to write to file.

This strategy pattern design also allows for easier implementation of further compression schemes that may be needed.

7. Limitations

Over the course of the development of this program, from the inception to the final production, there was little in the way of the main functionality of the application. Of the few issues that were experienced, most of the problems stemmed from basic design decisions, and not from problems in the code itself. One possible change that could have been implemented would have been to have multithreaded components in an attempt at efficiency in search queries.

8. Works Cited

Black, P. E., 2017. *Inverted Index.* [Online]
Available at:
https://www.nist.gov/dads/HTML/invertedIndex.html
[Accessed 4 August 2019].