Implementation of a Simple Web Search Engine

Tianhao Li tl1924@nyu.edu

1. Introduction

This application is a web search engine bases on Common Crawl data sets which is an open repository of web crawl data. This web search engine is running based on the inverted index built from the crawl data. It supports the conjunctive and disjunctive query.

After starting up, users can type in queries they want to search. The engine would response 10 URL results companying with this webpage's BM25 score. The result list is sorted in descending order according to the scores. In conjunctive mode, the engine would only return webpages containing all of the query terms. While in disjunctive mode, pages containing any of these terms would be returned to the users.

2. Data sets

All Common Crawl data is in WARC file format and also in .gz format. WET files which store extracted plaintext from the data stored in the corresponding WARC file are used, in order to reduce parsing work load. Each WET file contains file metadata at the head and around 53, 000 records. Each record also contains metadata and. The WARC metadata contains various details, including the URL and the length of the plaintext data, with the plaintext data following immediately afterwards.

These data sets also contain a small portion of web pages in languages that are not in Latin script such as Chinese, Japanese. These pages will be filtered by regular expression. Thus, the remaining words are all composed by [a-z1-9+].

In this search engine, totally 100 WET files are used. After parsing the data, it obtained 5.3 millions of URLs.

In the provided queries data sets, there are totally three query sets containing respectively 10 thousand, 20 thousand and 40 thousand queries. The largest one is used to warm up the search engine cache. The smaller ones are used to test the search engine.

3.Internal design

3.1 Redesign of inverted index building

Based on previous work, several changes have been made on the index building program.

3.1.1 Parser

The Parser class is for parsing the WET files. *parseWET()* and *parseRecords*() are the most important two functions. *parseWET*() is used to read records in a WET file and to write the posting returned from *parseRecords*().

In previous work, the data are plain text in *gzip* format. When reading the files, an index file is needed to seek a whole page. All pages are in html file format. The parser was designed to for html files. It would assign a document ID to each of these files, then extract the words between the opening and closing html tags, and write word with doc ID and frequencies of the word in this document. Also, write doc ID and URL to *url_table* file.

However, in this project, as stated in previous section, the payload of each WET data record, is already parsed to plain text. Moreover, the plain text may contain other languages' words or characters. In order to obtain all single words or characters sequence, in this project, regular expression is used to filter out those non-Latin script. Here the Parser would consider two words connected by underscore as one term. Same as previous, write the posting to the intermediate posting files. In the *url_table* file, the length of the document would also be written, which can be done easily through reading the metadata of the record. The length is important for calculating the BM25 score later.

3.1.2 IndexBuilder

The *IndexBuilder* class is for building index from the intermediate posting files generated from *Parser*. The index file contains document IDs and frequencies of all terms in this collection. It is in binary and compressed by encoding algorithm. Also, IndexBuilder would generate lexicon file as an auxiliary lookup file, a hash table storing for each distinct word *t* in the data set, a pointer to the start of its inverted list, and *f_t*, the number of docs containing *t*.
In the old version, the inverted index is in the format of

docID1 freq1 docID2 freq2 … docIDn freqn,

in which the *docID* are sorted in ascending order. And the *docID* is actually the difference between continuous two numbers, except the first one of each list. This reduced the value of this numbers, which would improve the compression performance.

However, in order to improve the performance of the search engine. The inverted index lists are compressed in blocks. Also each chunk can be decompressed individually. This design enable *QueryProcessor* jump forward without decompress the whole list, but can skip the whole block if the last also the largest *docID* in this block is smaller than the target ID.

To implement blocks compression, some extra auxiliary data are needed. After consideration the tradeoff between space and time, the format of each inverted index list is designed as below:

numOfDocs, [lastDocID1, …, lastDocIDn], [sizeOfBlock1, …, sizeOfBlockn], [docID1, …, docID128], [freq1, …, freq128], ….

in which each block contains 128 *docIDs* or *frequencies.* The *docID* and *frequencies* are compressed, and the auxiliary table is not in compressed format. Note that, the last block may contain less than 128 *docID* and *freq*. It will also be treated as a whole block.

In the lexicon, each line contains the term and the bytes pointer in the inverted index file so that search engine can fetch whole index directly from the file.

3.1.3 VarBytesCoder

This class is for compressing the payload of inverted index. The algorithm used is variable bytes encoding. After testing, the compression factor is about 2 on the inverted index.

In the previous version, the numbers in the list is compressed individually one by one since the *docID* is mixed with *freq*. In this version, after computing differences, the *docID* list and *freq* list are divided into 128 each block. The *VarBytesCoder* would compress the block together.

The var-bytes comporession algorithm is as follow,
If num < 128, use one byte, highest bit set to 0
If num < 128^2 = 16384, use two bytes(first has highest bit 1, others 0)
If num < 128^3, use three bytes, and so on.

3.2 Design for query processing

3.2.1 Index

In order to manage the inverted index of each word easily, a new class Index is designed to represent inverted index and auxiliary data, also to store decoded data.

Inside Index class, there are several fields.

term: the word of this index.
ft: number of documents containing this term.
numOfBlocks: number of blocks in this Index.
lastDocID: integer array including the last docID of each block.
blockSize: integer array including the block size in bytes.
payload: byte array containing all block data.
position: an index of block indicating which one is the current block.
blockPointer: a byte pointer pointing to the beginning of current block's docID part.
freqPointer: a byte pointer pointing to the beginning of the current block's freq list part.
decodedDocID: a hash table that contains all decoded docID block of this instance.

decodedFreq: a hash table that contains all decoded freq list block of this instance.

Upon each time the constructor of Index class is called to create an instance of Index class. The constructor would get a *Bytebuffer* and a *String* of term. The *Bytebuffer* contains the corresponding part of this term. The header of the *Bytebuffer* is the metadata part of the index. The *constructor* reads the first integer of, and set it as *ft*. According to *ft*, *numOfBlocks* would also be calculated since the doc numbers of the docID list is fixed as 128. Then the later integers until the *numOfBlocks* one would be the *lastDocID* list. Sequentially, the later *numOfBlocks* integers copied to *blockSize* list. Finally the rest of this *Bytebuffer* is the inverted index blocks, which is pointed by *payload*.

After reading all of inverted index data of this term, other fields of this instance are created and initialized based on these data. The initial *position* would be set to zero for every newly created instance as the index of two arrays *lastDocID* and *blockSize*. Also, it is corresponding to *blockPointer* and *freqPointer*, which would respectively point to the beginning of first docID and freq list blocks. The *decodedDocID* and *decodedFreq* would be two empty *HashMap.*

There are several important functions in Index class.

skipNext(): This function would be called when the *QueryProcessor* wants to loop through the inverted list of specific term. This function would judge whether it's possible or not to skip to next block. If there is at least on block after the current block, then it would move the *position, blockPointer, freqPointer* to the next one and return *true.* If current block is already the last block inside this Index's inverted index. Then the current position would be set to 0 and return *false.*

decodeCurrDocID(): This function decodes the docID of current block, and puts the docID into the decodedDocID container. If the docID of current block has already been decoded, then return it from the container directly. Otherwise, decodes the docID of current block, and recovers it to the real docID but not the difference between them. Finally, put the array into the container and return the decoded docID to the calling function.

decodeCurrFreq(): This function does the similar thing as *decodeCurrDocID().* Instead, it will decode the freq list of current block, and return it to the calling function.

getDecodedDocID(int did): This function simply returns the decoded docID list that contains a specific doc id.

reset(): This function resets all the pointers to the initial state, namely to the first element it was pointing to, and also clear the decoded lists containers which are *Hashmaps* .This function would be called before cache an Index of a term into memory to make sure that all Index in cache are fresh.

3.2.2 QueryProcessor

This class starts up the search engine and deals with queries input from users and returns the top 10 results to the users. This class supports two different kinds of queries search, conjunctive and disjunctive search. Basically, the conjunctive search implements an AND search among query terms, and disjunctive search implements a OR search among query term.

Inside Index class, there are several fields.

dAvg: average of documents length in the collection. It is computed when loading the URLs table into the memory. The length of each document is at the end of each line of URLs table.

urlTable: This is an array of Strings that holding the URLs table. The index of this array is corresponding to *docID* of this web page. This reduced the redundant information stored in the memory.

lexiconArray: This is an array containing all distinct words in the collection.

offsetArray: This is an array containing the offset(pointer) of the corresponding term in inverted index. The indices of *lexiconArray* and *offsetArray* are consistently kept corresponding.

indexCache: This is an *LRUCache* instance that dynamic caching Index into memory based on LRU policy.

heap: A TreeSet that hold the processed webpages' URLs and their BM25 scores. They are sorted based on their score. The size of this set would be kept 10 so that we will not waste spaces for all useless URLs.

There are several important functions in QueryProcessor class.

startUp(): This function would be called when the QueryProcessor class be used for the first time. It loads the *lexicon* data and *url_table* data into the memory. When reading the *lexicon*, the terms and the pointer of this term in inverted index are stored respectively into two arrays. And during the reading of *url_table*, the length of each document would be accumulated and then divided by the number of URLs bringing the average document length of the collections.

warmUp(): This function is for loading a small snippet of the inverted index by opening a list of query terms in a file. The files could be a trace of queries or a list or frequent words.

search(): This function is the first one touches the query. Firstly, it would do a word count to the query to omit the duplicate terms in the query. This would reduce the work load for *nextGEQ*() later. Depends on the flag of the query, this function would call *conjunctProcess()* and *disjunctProcess()* to process the query. After either of these function updates the results container *heap*, it would print the results and the process time out in the console.

conjunctProcess(): This is the process function for conjunctive query. Firstly, it would call *openList()* to open and construct the Index of each distinct term by calling binary search on the sorted array of all terms and then finding the corresponding offset based on the index in the array. Then, it would sort the list of Index in ascending order to make sure that the Index with the shortest docID list comes first. Then starts from the first doc ID of from the shortest inverted index in the array. Looping through the rest of the Index list, look up if any entries can be find in all of these inverted indices. When find any of the docID contains all terms in the query, open the freq lists of each term. Then compute the BM25 scores by calling the computing function. Then process the next doc id. After getting the score of specific doc, update the results container. Finally, cache those opened Indexes into the cache.

disjunctProcess(): This function process disjunctive queries. Different from conjucntProcess, it loops through every single words in the query. Each time, open the Index from the disks, then loops through all doc ids in the inverted list, compute the BM25 scores of each of them and updates them into the results container.

openList(): This function opens the inverted index in the disk. After get the word, it will do a binary search in the lexicon array. If found, with the same index in lexicon array, it looks up the offset array of the corresponding word. The length would be simply computed from the next term's offset. After get the offset and length, call *fetchPage()* to read the data from the disk and create an Index instance for this word.

nextGEQ(): This function is important for efficiently find the intersection of inverted index list for different words. With our block design for the inverted list, it enables us to *skip* block to block if no intersection doc ids have been found in this block. This would improve the speed of the process of looking for intersections.

3.2.3 LRUCache

This class takes control of the cache of the Index. The inverted index file is very large and is unable to be read into memory. However, we can cache some part of index into memory either at the beginning of the program or during queries coming in or both. This would reduce the time for catching index.

The cache can be in different types. One is static cache, it happens at the beginning of the program. By reading some queries or high frequency words, the static cache will be fed soon. Another type is dynamic cache, this cache happens during users querying in the system. After each time the QueryProcessor processed the query, it would cache the inverted index of terms in the query in to the cache.

But RAM space is limited, we cannot simply cache everything into the memory. Hence, we need to follow some policies to decide whether it should be cached or

not. The policy used in this search engine is LRU. LRU discards the least recently used items first. This algorithm requires keeping track of what used and when. In our design, it is implemented by using LinkedHashMap.

Inside Index class, there are three fields.

size: The current size of the LRUCache instance, it would be updated when put and remove.

maxSize: The maximum size of this LRUCache instance, a.k.a. capacity.

cache: A LinkedHashMap to keep the cache Index.

Functions:

set(): If the cache has already contains the Index of specific word, it would firstly remove it from the cache which is a queue. If not contains and the cache is already full, it would remove the first element in the queue. Finally, put the Index of the specific word in to the cache. This keeps the LRU policy be satisfied.

get(): Tries to get the Index of specific word. if it is not null, this function would call *set()* to remove it from the cache.

4. Users Interface

This web search engine is web accessible. The server is based on TomCat and Servlet library of Java. After launch the application, users can simply type in queries they want to search, and click on the corresponding button of “And” or “Or” which stands for conjunctive and disjunctive search respectively.

Also, users can run the program via Terminal, the server would prompt “Query:” to users. Users should type the query followed by “and” or “or” and separated by a space to the query.

Both way, after queries processed, the server would show the top 10 results’ URLs and BM25 scores to the users.

5. Performance

The search engines used 100 WET files and gained a 5 GB inverted index file with 5 million URLs and 38 million distinct terms.

After startup, it takes around one minute to read the URLs table and lexicon into the memory. For conjunctive queries without too long inverted list, it takes only hundreds milliseconds to process query. But for a bunch of top frequent words, it takes much longer time around 40 seconds. Usually, for disjunctive search, since it needs to search every single doc ID in the inverted list, it would take longer time.