**Assignment #3**

By Ziming Song

[zs2815@nyu.edu](mailto:zs2815@nyu.edu)

## **Introduction**

In this project, I built a search engine using C++, which can answer queries typed in by a user. It mainly contains two parts, index generator and query processor. In index generator, the program builds inverted index, lexicon and document table in compressed form. In query generator, the program uses these three structures to answer user’s queries, returns ranked list of scores, URLs related to the query keywords and snippets around them.

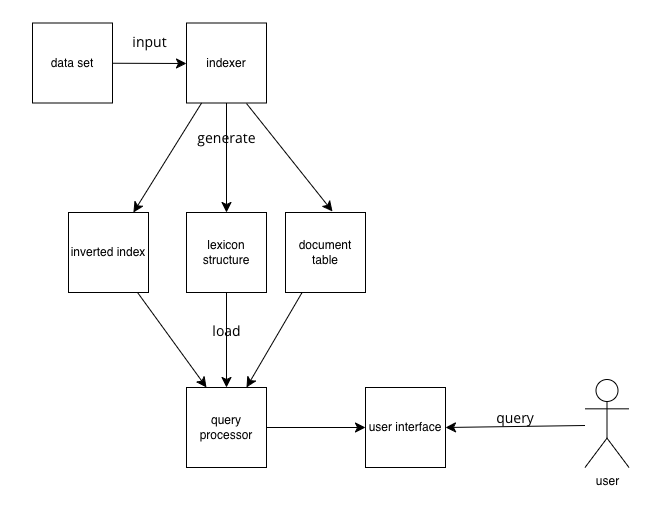
****

Figure overview

## **Module & Essential Function**

This program contains several modules, which are listed as follows. In each modules, I introduce its essential functions. Their relationships are shown in figure 2.

**DataLoader**: The top module which control other modules. It can call other modules to build inverted index and answer queries.

* **ReadData**: call module DocTable to build document table.
* **extractContent**: called by ReadData, extract specific string in a document in input dataset.
* **mergeIndexToOne**: merge several temporary inverted index into one uncompressed inverted index.
* **BuildLexicon**: call module Lexicon to build Lexicon Structure.
* **QueryLoop**: User Interface. Process use input and print output by calling result module.
* **TAATQuery**: called by QueryLoop, using TAAT(Term-At-A-Time) algorithm to get query answers.
* **updateScoreHash**: called when using conjunctive query, update the score of documents already in score hashtable.
* **findTopKscores**: find top K scores in score hashtable or score array and call module Result to record the result.
* **findDocID**: find whether docID in a term’s Inverted Index. Called by function updateScoreHash.
* **decodeBlocks**, **decodeBlock**, **decodeChunk**: decodeBlocks is used to decode blocks of a term. decodeBlock called by decodeBlocks to decode a block. decodeChunk called by decodeBlocks to uncompress a docID chunk or a frequency chunk.
* **openList**: uncompress and read metadata of a block.

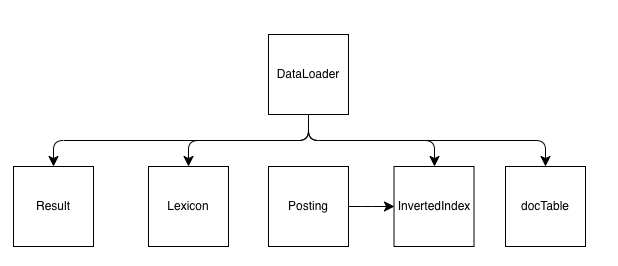


Figure module relationships

**DocTable**: Build and reload document table.

* **calcAvgDataLen**: calculate the average data length of all documents.
* **add**: add a document information in document table.
* **Write**: write document table from main memory to disk file.
* **LoadDocTable**: read document table from a disk file to main memory.

**Lexicon**: Build and reload Lexicon Structure.

* **calcDocNum**: Count the number of occurrences of a term in different documents.
* **Build**: generate compressed and block-based inverted index. Build lexicon structure.
* **Insert**: insert a term information in lexicon structure.
* **Write**: write lexicon structure from main memory to disk file.
* **LoadLexiconList**: read lexicon structure from a disk file to main memory.

**InvertedIndex**: Build temporary inverted index.

* **Clear**: Clear temporary inverted index.
* **Write**: Write temporary inverted index into a temporary disk file.
* **Insert**: Insert a posting in temporary inverted index.

**Posting**: Store and print information of posting. A posting is consisted of a term and the frequency of the term in a specific document.

**Result/ResultList**: Record top K result information, generate snippets, output results.

* **Clear**: Clear result list.
* **Insert**: Insert a result in result list.
* **FindSnippets**: find snippets of results in result list.
* **Print**: print information of results from highest score to the lowest score.

## **Essential Design & Algorithm**

**Document Table Design**

Document table is a list of information of several documents. Each element of the list contains its document ID, document Number (provided by original dataset), data length (document size), number of distinct words, URL.

*{docID0, docNO0, dataLen0, word\_num0, URL0}*

*{docID1, docNO1, dataLe10, word\_num1, URL1}*

*…*

*{docIDn-1, docNOn-1, dataLenn-1, word\_numn-1, URLn-1}*

**Document Table Generation**

This process need module DataLoader, DocTable. The main function to implement document table generation is *DataLoader::ReadAll*.

Use C++ zlib library, read dataset *msmarco-docs.trec.gz* in compressed format. Uncompress the dataset in buffer. Assign docIDs based on traversal order.Find documents one by one according to tags (<DOC>,</DOC>) in buffer and extract information in each document.

Find URL in first line of document TEXT. Record and calculate different words using hashtable. Using string in C++ to recognize UNIX characters.

During the document table generation, also record each term(word) and its corresponding frequency in document. Write those postings in module InvertedIndex.

**Inverted Index Design & Layout Design**

In order to improve speed and facilitate debugging, I compress binary inverted index and write it in block format after all temporary inverted index files merged.

For temporary inverted index, it is implemented using ordered map. The key of ordered map is word(term), the value of the word is a list of postings (*{docID, freq}*).

*word0: {docID0, freq0}, {docID1, freq1}, …, {docIDn-1, freqn-1}*

*word1: {docID0, freq0}, {docID1, freq1}, …, {docIDn’-1, freqn’-1}*

*…*

*wordn-1: {docID0, freq0}, {docID1, freq1}, …, {docIDn\*-1, freqn\*-1}*

For the final inverted index, it is compressed and block based. The structure is shown in figure 3. Each Term have one Inverted List. An Inverted List is stored in 1-n blocks. A block only contain one term’s full or partial inverted list.

Each blocks is less than 64KB. Each block contains metadata, several docID chunks, several Freq chunks. A docID chunk contain 64 doc IDs. The first record in a docID chunk is original docID, other records in the docID chunk are the increment of previous docID. Similarly, a Freq chunk contain 64 frequencies.

The program use Varbyte to compress each record in docID chunks and Freq chunks. So, the size of docID chunks and Freq chunks are not fixed.

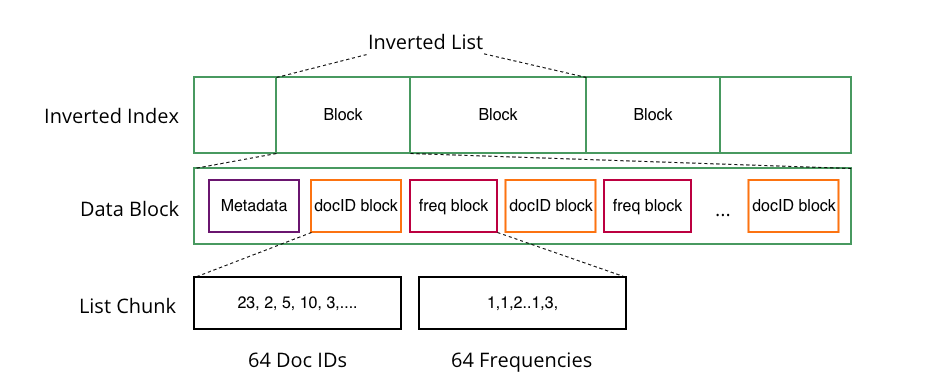


Figure Inverted Index Layout

As for metadata, it contains four parts. *Metadata\_size* record the size of each three lists in this block. *last\_docID\_list* records the last docID in each docID chunk. *docID\_size\_list* records the size of each docID chunk. *freq\_size\_list* records the size of each freq chunk. The length of *last\_docID\_list*, *docID\_size\_list*, *freq\_size\_list* is the same, that is, *Metadata\_size.* Metadata is uncompressed.

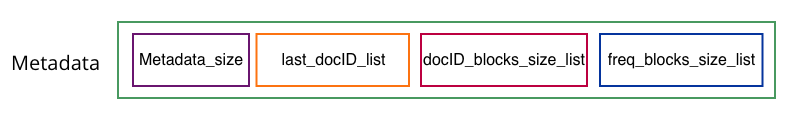


Figure metadata

**Inverted Index Generation**

The Inverted Index Generation need three steps.

1. **Index Setup**. Write temporary Inverted Index in disk file.

This process is carried out simultaneously with document table generation. This process needs module DataLoader, Posting and InvertedIndex. The functions contain *DataLoader::ReadAll*, *InvertedIndex::Write*, *InvertedIndex::Insert*.

1. The data will be read into buffer of size 50kb. The program will parse each document’s information according to ‘.trec’ format.
2. For each distinct word in a document, calculate its frequency. Get a SortedPosting List and insert each posting into InvertedIndex one by one.
3. When the InvertedIndex size bigger than FILE\_INDEX\_CHUNK, write InvertedIndex in a bit/text format file in INDEX\_FILE\_FOLDER\_PATH, then clear the InvertedIndex. After parsing all documents, write remaining InvertedIndex in file.
4. **Merge.** Merge Inverted Index files into one disk file.

This process needs module DataLoader, InvertedIndex. DataLoader call *mergeIndexToOne* function, using merge-sort algorithm to merge files in INDEX\_FILE\_FOLDER\_PATH.

1. Because docIDs are assigned according to the order the documents are parsed and postings in InvertedIndex are sorted, words in InvertedIndex List and docIDs in inverted list in each unmerged file in INDEX\_FILE\_FOLDER\_PATH are in ascending order.
2. If the number of unmerged files is greater than 1, merge two of them in order and write the sorted files to a new file under INDEX\_FILE\_FOLDER\_PATH. The two original files are marked as merged, the new generated file is marked as unmerged.

Details about merging file1 and file2 to dst\_file:

I. Pointer p1 and p2 point to reading position of file1 and file2 respectively. Get Inverted List index1, index2 in file1 and file2 respectively.

II. Get word1, word2 of file1 and file2 respectively. If word1<word2, write index1 in dst\_file. If word1>word2, write index2 in dst\_file. If word1=word2, merge index1 and index2 in dst\_file. For merging index1 and index2, compare the first docID of index1 and index2, the items with smaller docID should be put first.

III. Repeat II until p1 of file1 or p2 of file 2 move to the end of corresponding file.

IV. Write the remaining file1 or file2 to dst\_file.

1. Repeat b. until unmerged files no greater than 1.
2. **Compress and write binary inverted index file using Inverted Index layout.**

This process is carried out simultaneously with lexicon structure generation. This process needs module DataLoader, Lexicon. The main function is *Lexicon::Build, Lexicon::WriteBlocks*.

1. In the function, the program read final merged file. For each term, compress its postings’ docID and frequency. Then divide compressed docIDs into groups. Each groups contain 64 docIDs, except the last group. The last group can have docIDs less than 64. And divide the frequencies as the same way. Each of the divided group is called chunk.
2. Allocate docID chunks and frequencies chunks, make sure each block’s size is less than 64KB. Calculate metadata of each block.
3. Write blocks of each term into final index file

Repeat a.b.c. until all terms are written.

**Index Compression**

As previous mentioned, the program compress docIDs and frequencies in each chunk in step 3 of Inverted Index Generation. The function is *varbyte\_encode.*

For docIDs compression, because docIDs are increasing. I only record the first original docID in the chunk, others I record the docID’s increment of its previous docID. Then I use varbyte to compress. For frequencies compression, I use varbyte compression.

Varbyte(VB) compression:

Varbyte is a simple byte-oriented method for encoding data. It encodes numbers as follows:

- if n < 128, use one byte (highest bit set to 1)

- if n < 128\*128, use two bytes (first has highest bit 0, the other 1)

- if n < 128^3, then use three bytes

So highest bit 0 means the number continues in the next byte. We write bits from the lowest 7 bits to the highest 7 bits.

Suppose you want to encode the integer 150. In binary, 150 is represented as 10010110. Varbyte encoding breaks this binary representation into chunks of seven bits. The Varbyte encoding for 150 would be something like: 00010110 10000001(spaces added for clarity).

**Lexicon Structure**

Lexicon structure is a list of a term’s/word’s information in Inverted Index. It is designed using hash table. The key of the hash table is term. And the value is a tuple of this term’s information in Inverted Index. The information contains begin and end offset of the term’s inverted list in the final Inverted Index file (*begin\_pointer, end\_pointer*), how many documents does the term appear in (*docID\_nums*), how many blocks the inverted list have (*block\_nums*).

*word0: {begin\_pointer0, end\_pointer0, docID\_nums0, block\_nums0}*

*word1: {begin\_pointer1, end\_pointer1, docID\_nums1, block\_nums1}*

*…*

*wordn-1: {begin\_pointern-1, end\_pointern-1, docID\_numsn-1, block\_numsn-1}*

**Lexicon Build**

As previous mentioned, the Lexicon Structure is built during the step 3 of Inverted Index generation. This process needs module Lexicon. Function *Lexicon::Build* call function *Lexicon::Insert* to insert information in Lexicon hash table. Function *Lexicon::Build* record the offset before and after a term’s inverted list writing, and store them respectively in variable *begin\_pointer, end\_pointer.*

**User Interface**

User can see outputs of the program in terminal. The terminal outputs contains the time spent on each step in inverted index generation, the time spent on writing lexicon structure and document table to disk file, the query guidance, time spent on each query and search result.

User can also type queries in terminal. As for query process, it is a loop of search. The loop will not end until user type “exit” to exit searching. For a query in a loop, first, user need to type a query like “hello world”. Then user need to type its query type, conjunctive or disjunctive. The program will process the query and the output result in terminal.

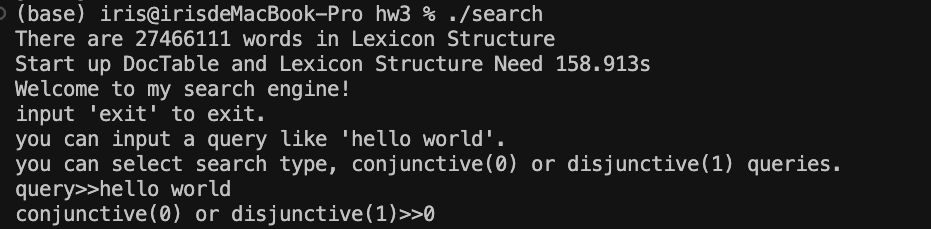


Figure 5 user interface when query

**Query Process**

According to the query and query type, query processor will follow these steps to give results.

1. Split query, generate word list.
2. According to query type, use different algorithm.
3. Find top K result, write them in module Result.
4. Snippets generation of top K results.
5. Output result.

Here are two types of queries, conjunctive and disjunctive. Conjunctive queries involve multiple conditions connected by "AND" operators, where all conditions must be satisfied for a result to be retrieved. Disjunctive queries use "OR" operators and return results that meet at least one of the specified conditions.

TAAT is an information retrieval technique that retrieves documents by evaluating each query term separately and combining their results afterward. For different types, the program uses different algorthim, but both based on TAAT(term-at-a-time) query processing.

**Conjunctive Query**

a. process one inverted list of a term at a time, from the shortest list.

b. initialize an empty hash table. I use ordered map in C++, which can make sure docIDs in ascending order.

c. for each posting in shortest list, create a hash entry with the docID as key and the posting’s impact score as value.

d. for each posting in the other lists, check if there is an entry in the hash table; if yes, add term contribution to score

e. output results with K highest values that contained all terms using heap

Details about step d:

At the beginning, the score hashtable *sh* is initialized using shortest term *t1*. Begin offset (*beginp*) of Inverted Index filepointed to the beginning of inverted list of the term *t2*. For each docID in *t2*’s inverted list, call function *DataLoader::findDocID* to check whether this docID is in *sh*, if so, add extra score contributed by *t2* to *sh*[docID].

Function *DataLoader::findDocID* can return whether a *docID* in a term’s inverted list from begin offset (*beginp*) of Inverted Index file. To speed up, I add *nextdocID,* which means the next bigger docID in ordered hash table. *beginp always point to a blocks’s begin. findDocID* will read metadata from the *beginp* using function *openList*, traverse last\_docID\_list to find whether last\_docID is bigger than *docID,* if not, jump this block and move *beginp*. If last\_docID is bigger than *docID*, decode the *docID* chunk and find whether the *docID* is in this chunk. And if the docID we traverse is equal or bigger than *nextdocID,* return false immediately.

*DataLoader::findDocID* will change *beginp*’s value, so we don’t need to start searching a docID at the beginning of an inverted list again and again, speeding up searching time.

**Disjunctive Query**

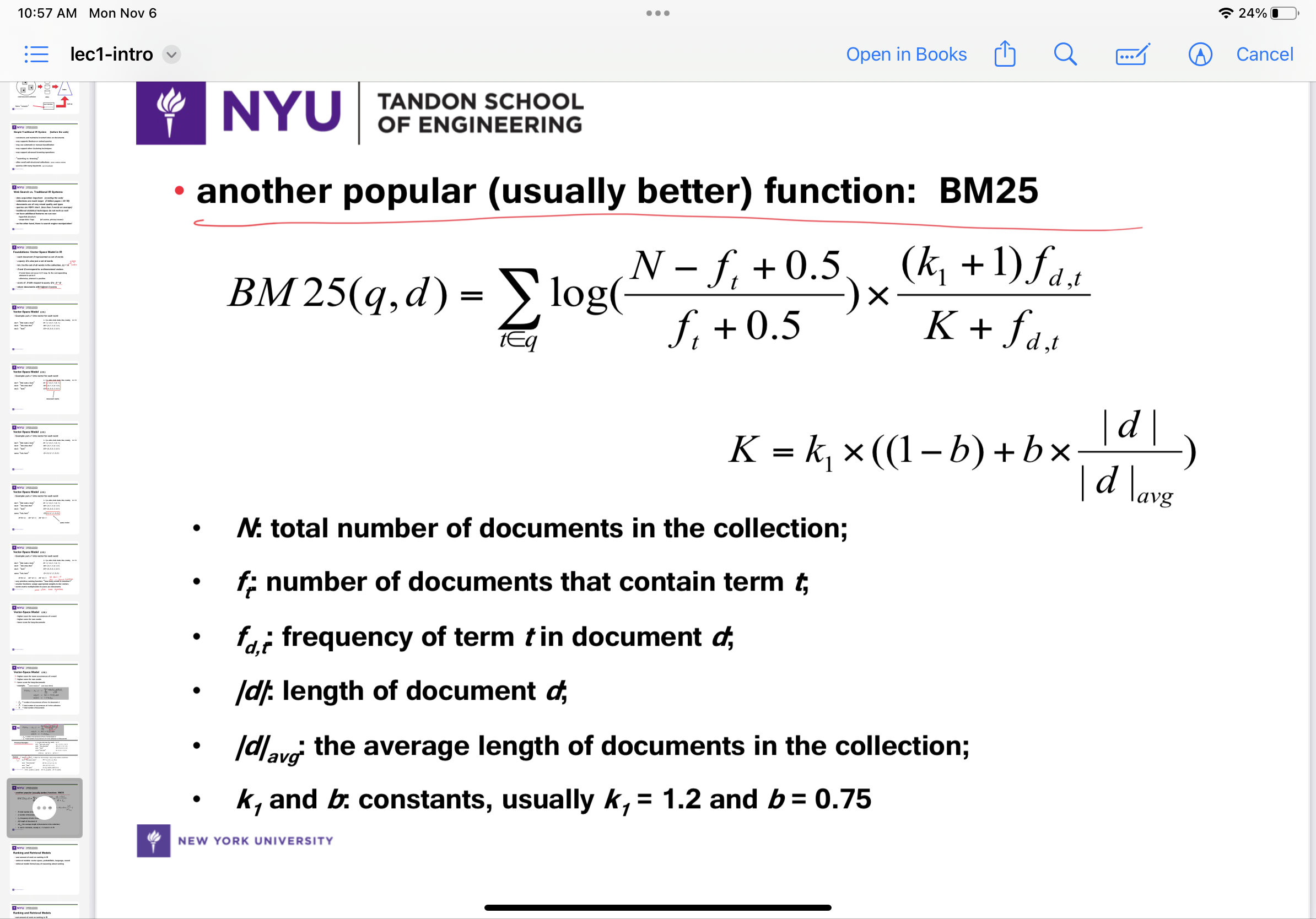
a. Initialize .Use a dense array for score accumulation. One entry per docID in the collection. For each query, initialize array to all zero.

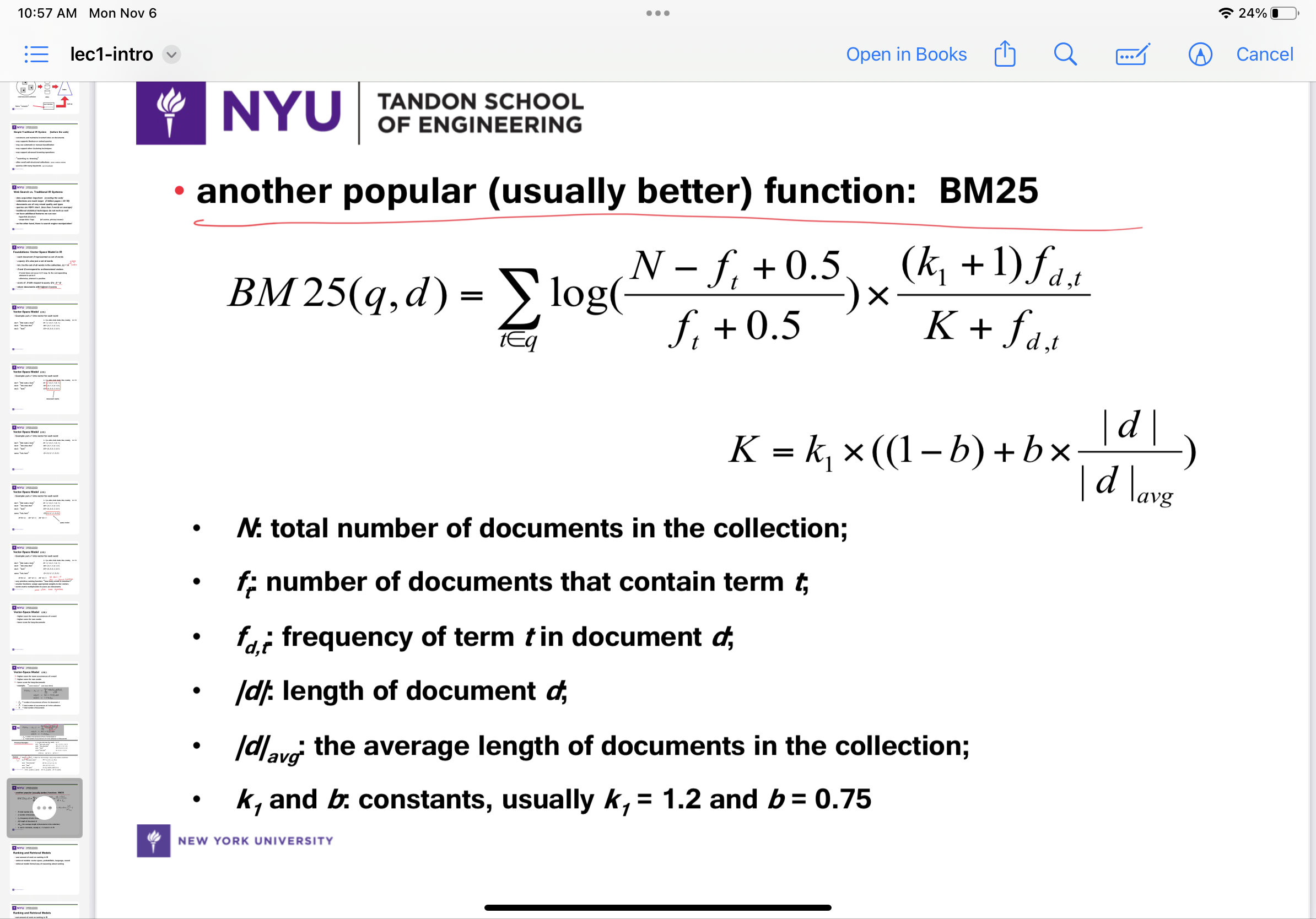
b. Traverse/Accumulate. Go through each term’s inverted lists and add impacts

c. Aggregate. Go through array again to find top K result using heap.

**Score Function**

The program use BM25 for score. The function is *BM25\_t\_q*. It will give a score based on term, docID and its frequency. The BM25 function and its parameters are shown as follows.





**Inverted List Reading & Decompression**

As mentioned, inverted list are compressed and compressed.

To read a term’s inverted list. The top function is *decodeBlocks*. Here are few steps in *decodeBlocks* function.

1. Get block numbers, begin offset and end offset from Lexicon Structure
2. Call function *decodeBlock* to decode a block one by one.
   1. read metadata
   2. call function *decodeChunk* to decode docID blocks and frequency block according information provided by metadata. The function *decodeChunk* is to decompress code of varbyte compression and return a list of original uint32 type value. The return list size is not bigger than 64 (posting numbers in a chunk).
   3. update begin offset.

**Result Design**

Result List is a list of information of searching results. Each element of the list contains its document ID, URL, BM25 score, snippets(nearly 100 characters) around query key.

*{docID0, URL0*,*score0, snippets0}*

*{docID1, URL1*,*score1, snippets1}*

*…*

*{docIDn-1, URLn-1*,*scoren-1, snippetsn-1}*

**Snippets generation**

The snippets generation occurs after finding top K results. Here are few steps to generate snippets.

1. Traverse result list, add top K result’s docID in an array *doc\_array*.
2. Sort *doc\_array* in ascending order.
3. Read dataset again and find docs which docID is in *doc\_array*.

Since both docID of dataset and *doc\_array* are in ascending order, I use two pointers to find documents. In this way, I only need to traverse dataset for one time.

When a docID is found, extract the doc’s text and find word by word to check if this word in query’s word\_list. If so, add the next and after 50 characters as snippets text. Suppose the length of the word found in word list is *lw*. The length of snippets is 100 + *lw.*

## **Workflow**

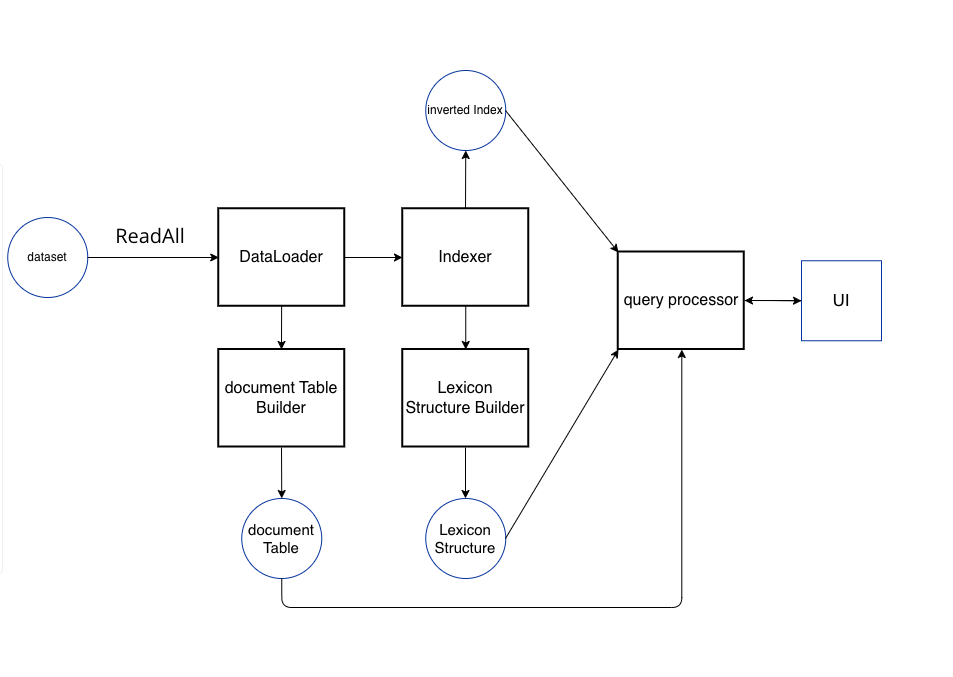


Figure workflow

## **How to Run this program**

This program using makefile and C++. You have to install and configure cmake first.

1. change “config.h” to configuration filepath, folder path. The meaning of config

|  |  |
| --- | --- |
| parameter | meaning |
| DATA\_SOURCE\_PATH | File path of “\*.trec.gz” |
| INDEX\_FILE\_FOLDER\_PATH | Folder path you want to keep temporary Index files |
| FINAL\_INDEX\_PATH | File path of final merged Inverted Index file |
| LEXICON\_PATH | File path of Lexicon file |
| DOC\_TABLE\_PATH | File path of document table file(or page table file) |
| FILEMODE | Write file in bit format or text format. Value should be set to 1(bit) or 0(text). |
| IS\_DEBUG | Indicate whether output debug information in terminal. Value should be set to 1 or 0. |
| IS\_INDEX | Indicate whether reindex input trec data. Value should be set to 1 or 0. If value is 0, it will merge files in INDEX\_FILE\_FOLDER\_PATH |
| IS\_MERGE | Indicate whether merge files in INDEX\_FILE\_FOLDER\_PATH. Value should be set to 1 or 0. |
| IS\_WRITE\_PAGE | Indicate whether write page table into file. Value should be set to 1 or 0. |
| IS\_WRITE\_LEXICON | Indicate whether Lexicon Structure into file. Value should be set to 1 or 0. |
| IS\_DELETE\_TEMP | Indicate whether delete temporary Index Inverted file. Value should be set to 1 or 0. |
| IS\_BUILD | Indicate whether build compressed and block-based Inverted Index file. Value should be set to 1 or 0. |
| IS\_RELOAD | Indicate whether reload document table and lexicon structure in disk file. Value should be set to 1 or 0. |
| IS\_QUERY | Indicate whether allow user query. Value should be set to 1 or 0. |

1. In terminal, type following command lines:

$ make

$ ./search

## **Result**

**Index Generation Time**

Build Postings and temporary Inverted Index Need 11835.2s

Merge Inverted Index Need 4689.11s

Write Lexicon Structure Need 65.8997s

Total time is 123260.21s

Total time is about 4.61h. If use unix sort to build temporary Inverted Index, total time is about 9793.52s(2.72h).

**Inverted Index Size**

Before compress, the temporary index is 13.62 GB. The final Inverted Index file size is 3.93GB.

**Start up Time**

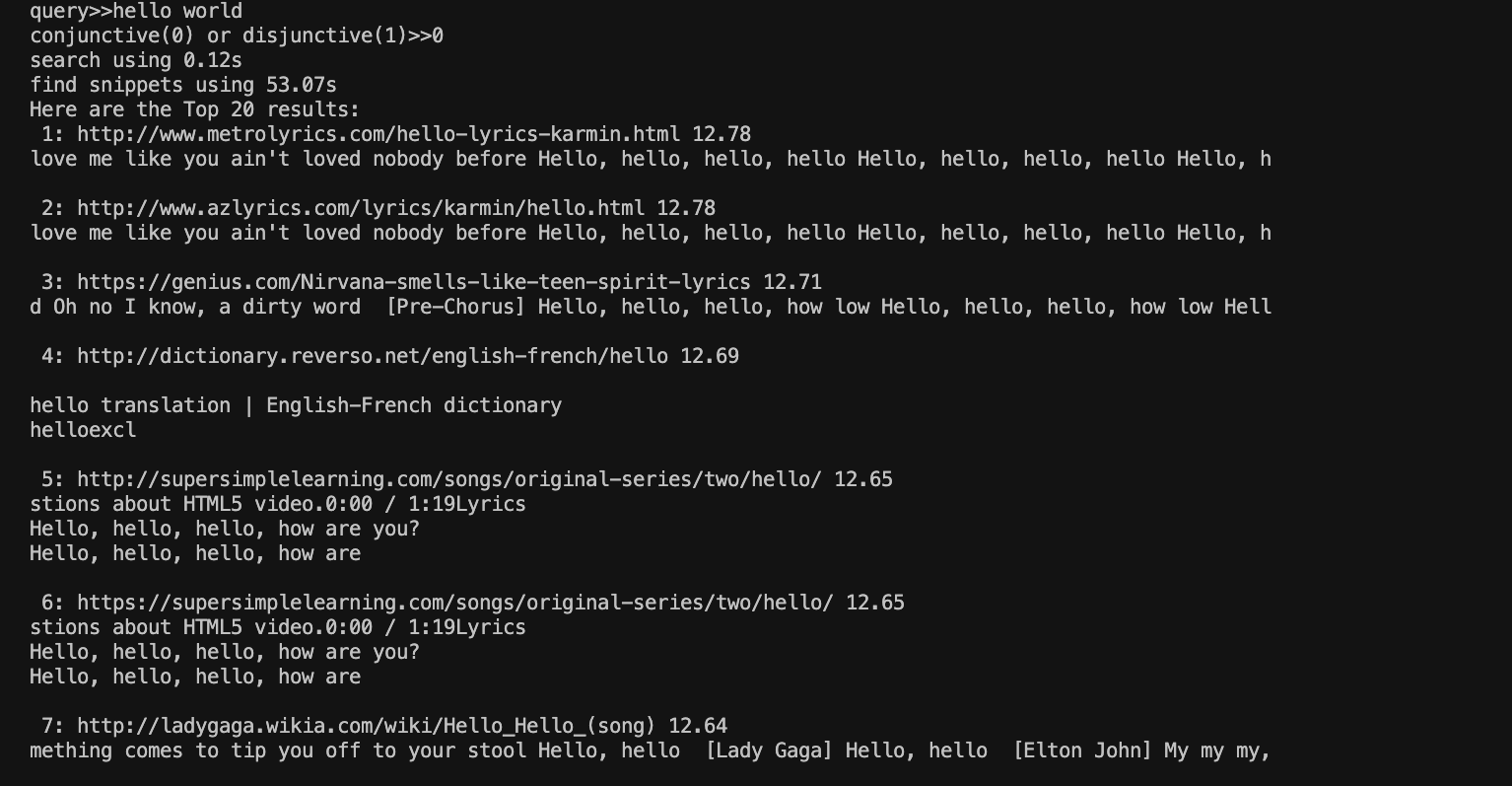
Start up DocTable and Lexicon Structure Need 158.228s

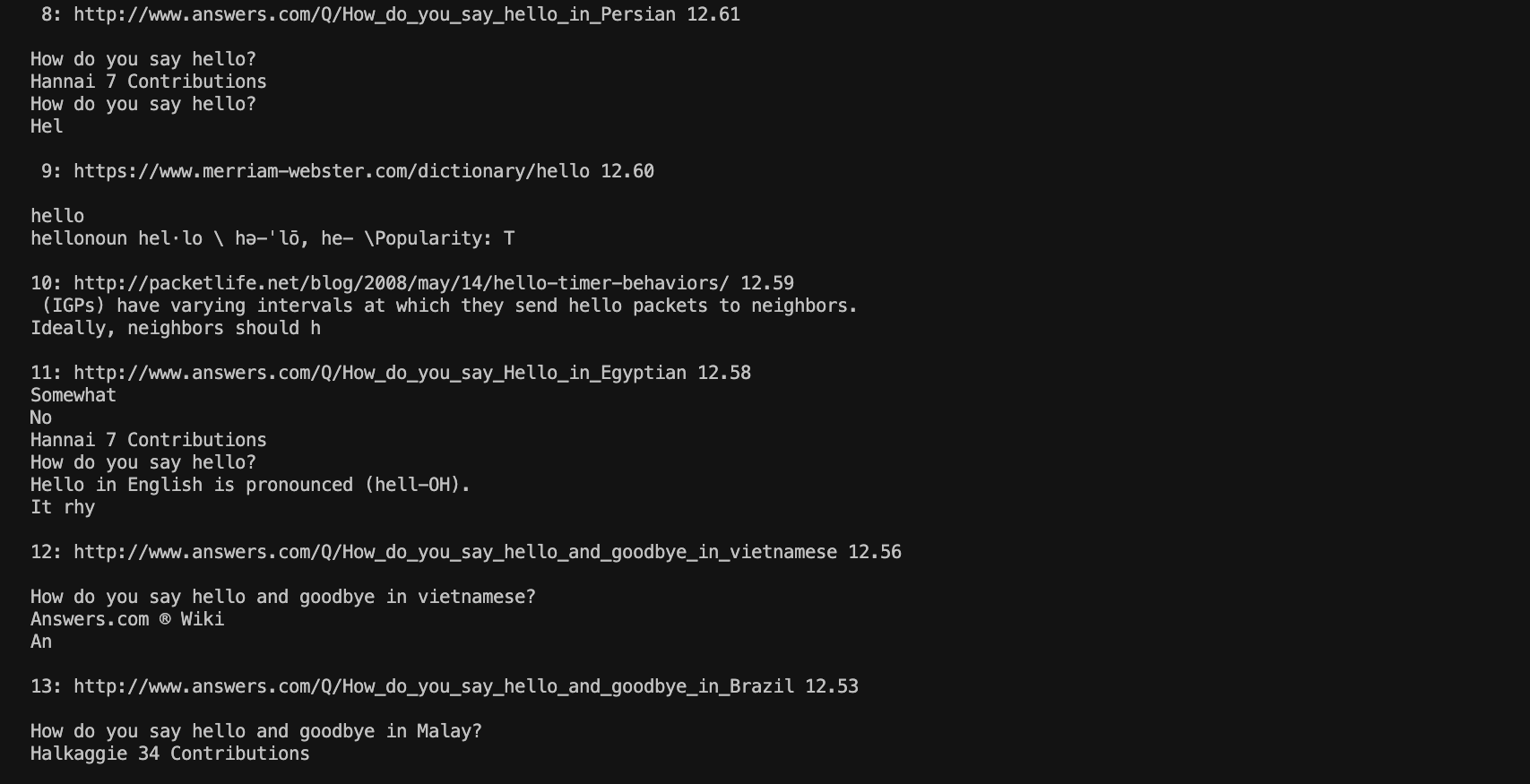
**Query Time and Result**

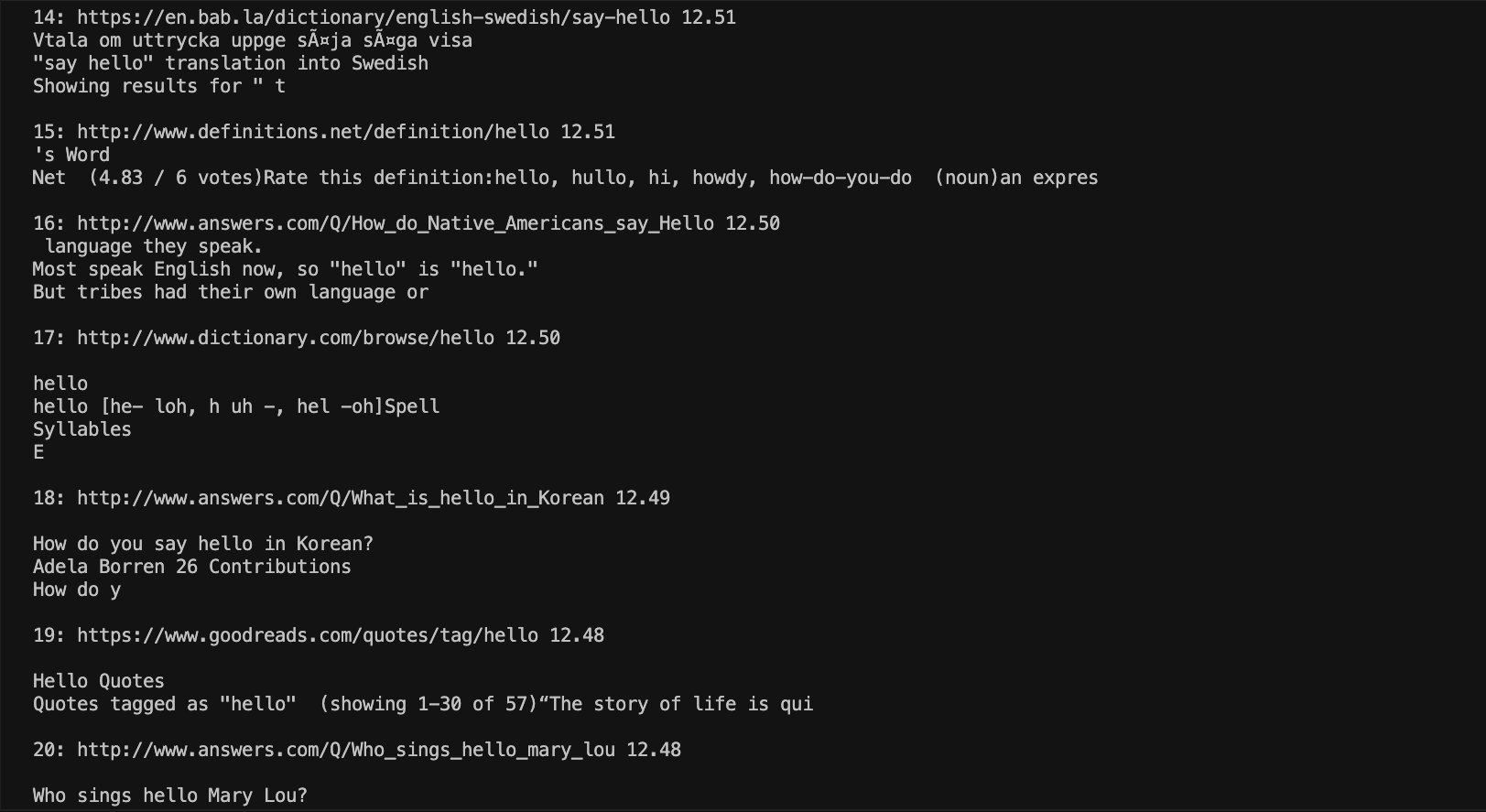
On the dataset, for a normal conjunctive query, search time is around 0.1s; for a normal disjunctive query, search time is around 0.1-1 s.

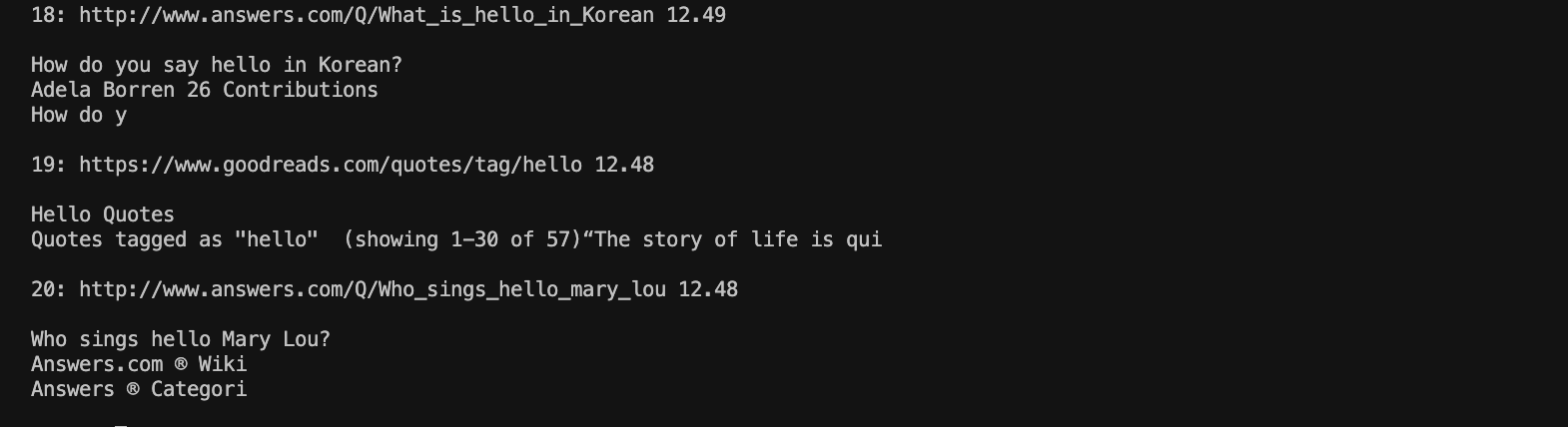
The time to find sippets is about 50s.

Here is a conjunctive query result. The red box marks URL, score and snippets of a result. The results are printed from highest score to the lowest score.

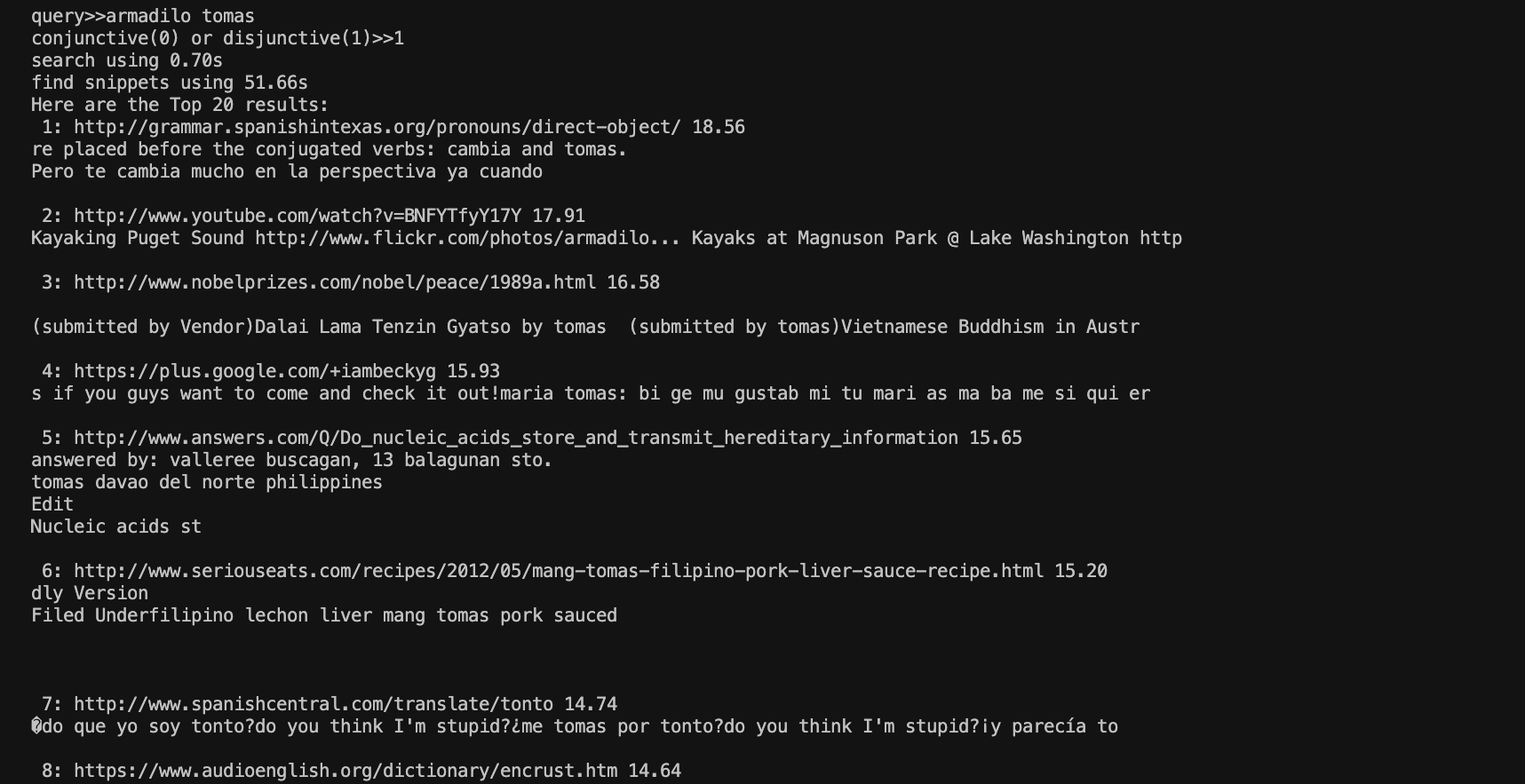


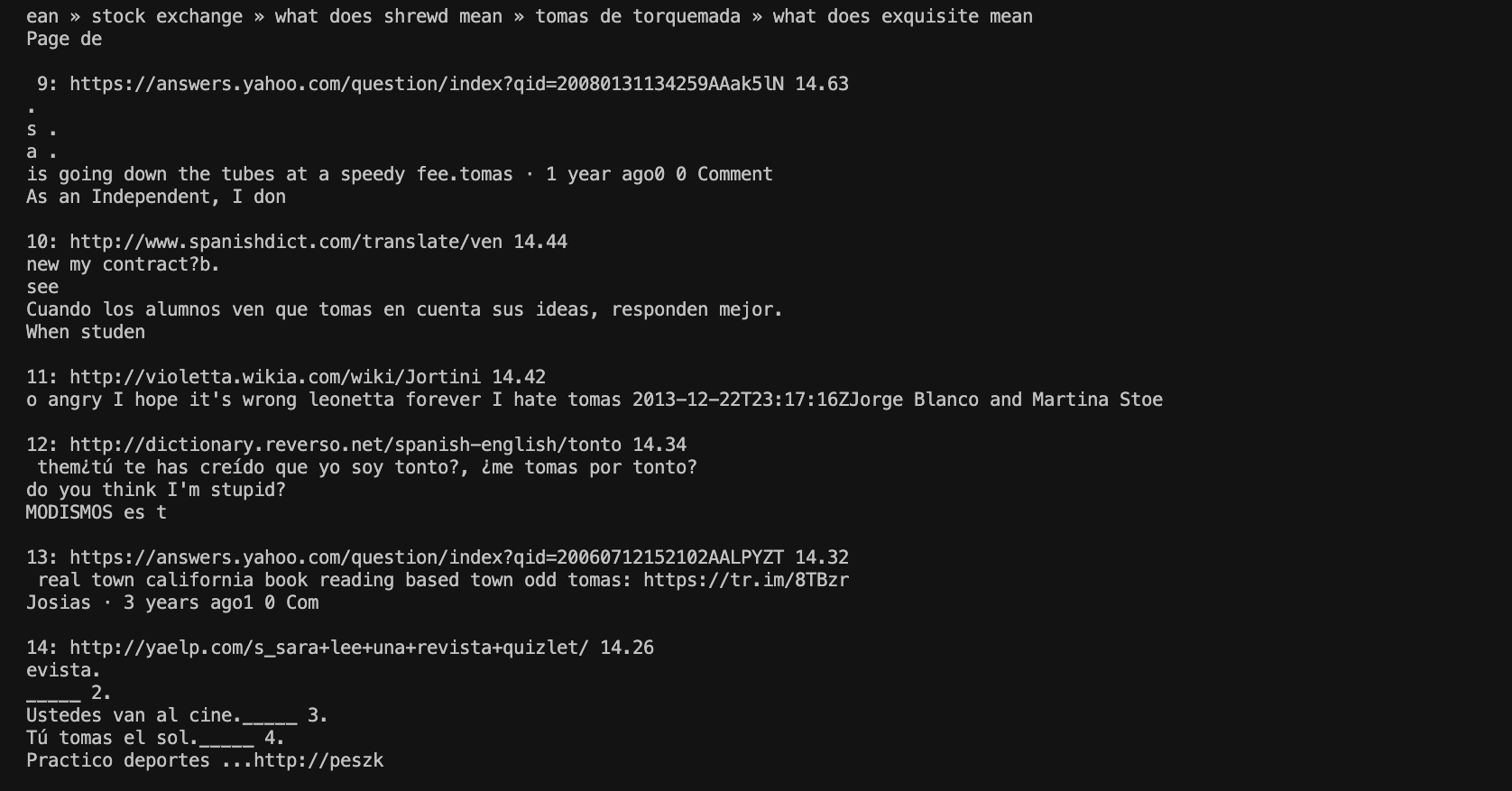


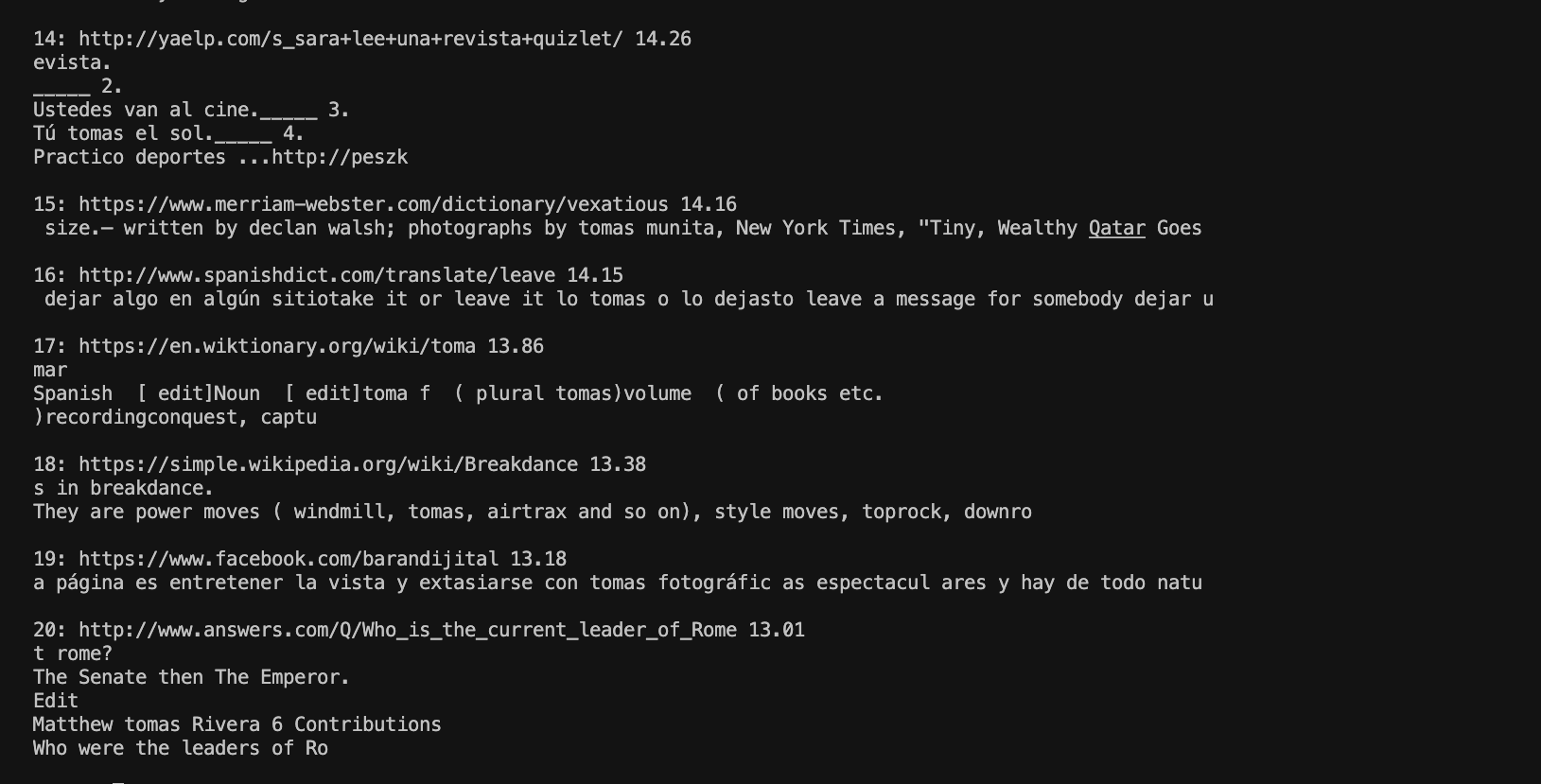




Here is a disjunctive query result.







## **Limitation**

Time spent on query is acceptable. But time spent on snippets generation is long.

## **Reference**

[1] slides in class