CS6913: Assignment #3

Hang Zhang
hz2447@nyu.edu

New York University — November 8, 2021

Contents

1 Introduction

This project is to parse the web pages, extract terms(words) from the pages and build index file for them. The whole process can be divided into four parts:

(1) Parse: Read and parse the web pages, generate several sorted intermediate posting files.

(2) Merge: Merge the sorted posting files into one sorted postings file.

(3) Build index: Read the sorted posting file, build the final binary compressed index file.

(4) Query: Given a query composed by several words, return the top 10 pages with highest BM25 score and generate snippets for them.

2 Main Implementation

2.1 Parse

The first step is to parse the crawled web pages and generate intermediate postings. Each posting is in (termid, document id, frequency) format. Since there are a great number of crawled web pages(3.2M in this paper), we can not cache all intermediate postings in memory. The solution is that we create a fixed-size output buffer to cache postings in memory, and sort them and write them into disk when the buffer is filled. So after this step, we have several files in disk and each stores the sorted postings.

2.2 Merge

The second step is to merge the sorted postings generated in the Parse process. Since memory is very limited, we must design an I/O efficient algorithm to merge the postings. The algorithm is that we create some fixed-size input buffers and one fixed-size output buffer in memory, each input buffer will read postings from one sorted posting file, and we do the process like merging k sorted list problem. Note that to achieve I/O efficiency, we will write the postings in the output buffer to disk when it is filled. Also, when an input buffer is empty, read another batch of postings from the assigned sorted posting file until it have read all postings from that file.

2.3 Build index

The index file is also very huge in size. So we can not cache all terms' inverted lists in memory. The solution is that we store all inverted lists in disk and record the offset and the length of each term's inverted list in lexicon. When we query a term, we just let the file pointer point to the start of the inverted list of that term then read the inverted list into memory. Note that we divide each inverted list into blocks and also store some metadata at the beginning of each inverted list for the blocks, which are the last document id and the size of each block, so we can quickly skip the block which does not contain the document id we are seeking. Also, all the data are compressed by VarByte method since most of numbers are not very big.

2.4 Query

2.4.1 Conjunctive Query

Conjunctive query means all the terms in the query must appear in the returned documents.
In this paper, I use "Document-At-A-Time" to implement conjunctive query. Enumerate each document id in the shortest inverted list, and find if all other inverted lists have that document id. If they all have that document id, calculate the BM25 score for that document and cache it in an array. Otherwise, check the next document id. Finally, we traverse the array and get the top 10 documents with highest score, then generate snippets for them.

2.4.2 Disjunctive Query

Disjunctive query means there is at least one term in the query appears in the returned documents.
So all terms share a common unordered map. For each term, traverse all document ids in its inverted list and calculate the BM25 score and add up the score in the hashmap. After traverse all terms' inverted lists, we traverse the unordered map and get the top 10 documents with highest score, then generate snippets for them.

3 Structure of this program

3.1 Tuple class

The "Tuple" class is defined in "info.hpp". This class has three member: termid, docid and freq. So one object of this class is one posting. It also overload the $<$ operator so the posting object with least term id will be at the top of priority queue, which will be used when merging. (If the termids are same, the posting with least document id will be at the top of priority queue.)

3.2 Parse process

The class Parser is defined in parse.cpp.

3.2.1 Constructor of Parser class

The main function of Parser constructor is to open the trec file by a ifstream object, define the delimiters which are used to split text in each document, new memory for the output buffer, create a directory ("./data") where the intermediate posting files will be saved.

3.2.2 Main function of Parser

parse() is the main function of Parser.
In the parse() function, the ifstream object will read the trec file line by line. There is a "flag" variable which denotes which field the current line belongs to.
If current line is "<DOC>", which means we enter a new document so we update the document id(++ operation).
If current line is "<TEXT>", flag will be set to 1, which means the next line will be the url of document.
If flag is 1, record the url of the current document id, set flag to 2, which means the next line will be the text of this document.
If flag is 2, concatenate the current line to a string variable called "content".
If current line is "</TEXT>", the text field ends, so set flag to 0. Then call extract() function to split words from the content(all lines of text are concatenated to a string called "content"), the extract() function return an unordered map where all the terms and their frequency are cached. Then pass the unordered map and document id to generate_postings() function to traverse the unordered map and generate postings which will be cached in the output buffer.

Algorithm 1 Parse process

Input: None
Output: None

```
1: function PARSE( )
2:     line ← emptyString
3:     content ← emptyString
4:     flag ← 0
5:     while getline(infile, line) do
6:         if flag == 1 then
7:             url ← line
8:             add url and infile.tellg() to document table
9:             flag ← 2
10:            continue
11:        end if
12:        if line == " </TEXT> " then
13:            Split content by delimiters, store <termid, frequency> as key-value pair in unordered_map
14:            Generate postings from the unordered_map to the output buffer
15:            flag ← 0
16:            content ← emptyString
17:            continue
18:        end if
19:        if flag == 2 then
20:            content ← content + line + ""
21:            continue
22:        end if
23:        if line == " <DOC> " then
24:            page_id ← page_id + 1
25:            continue
26:        end if
27:        if line == " <TEXT> " then
28:            flag ← 1
29:            continue
30:        end if
31:    end while
32:    return void
```

3.2.3 Split the text

void extract(string& content,unordered_map<int,int>&ms);
This function is to split the content(all lines of text from one document) by the delimiters. The library function I use to split is "strtok". The unordered_map is used to store the result of split, its key is the term id and the value is the frequency of that term.

3.2.4 Generate intermediate postings

int generate_postings(unordered_map<int,int>&ms);
This function is to generate postings to the output buffer. All the terms and their frequency are cached in unordered_map, so this function is to traverse the unordered_map ms, add the current document id with each key-value pair from the unordered_map to the output buffer. When adding a posting to the output buffer, it will check whether the buffer is filled, and if filled, sort the postings in the buffer and call write_out() function to write the postings in buffer to disk in binary format. This function will return the number of words in the current document and so we can add the number to the document table for the computation of BM25 during query process.

3.3 Merge process

The code is in merge.cpp.

3.3.1 Load class

The Load class is actually the input buffer class. The constructor is to open a file where intermediate postings are stored by an ifstream object in binary format, new memory for input buffer, then read the first batch of data from disk to buffer.
The get_data() function will return one posting from the input buffer. If the input buffer is empty, then call read_data() to read a new batch of data from disk by the ifstream object.
As mentioned above, the read_data() function will read a batch of data from disk to input buffer

3.3.2 Main function for merging

In the main() function, there's a queue which initially stores the paths of intermediate posing files generated when parsing. Each time it will get 16 paths from the queue and pass these paths to the merge_to_one_file() function, then push the path of merged posting file to the end of the queue. We keep doing this until the size of queue decrease to 1. Then write the path of the final merged posting file into a txt file since we will use it to build index.

Algorithm 2 Merge process

Input: vector<string>paths, containing the paths of sorted intermediate posting files
Output: a string which is the path of merged posting file

```
1:  function MAIN()
2:      while paths.size() > 1 do
3:          vector < string > tmp, newPath
4:          for i = 0 → paths.size() − 1 do
5:              append paths[i] to the end of tmp
6:              if tmp.size() == 16 then
7:                  call merge_to_one_file() to merge the files whose paths are stored in tmp
8:                  append the new merged file's path to the end of newPath
9:                  clear the tmp
10:             end if
11:         end for
12:         if tmp.size()! = 0 then
13:             merge the files whose paths are stored in tmp
14:             append the merged file's path to the end of newPath
15:             clear the tmp
16:         end if
17:         paths ← newPath
18:     end while
19:     return 0
20: end function=0
```

3.3.3 Merge several intermediate posting files into one file

string merge_to_one_file(vector<string>&paths, int round, int num, string dir);
The intermediate posting files' paths are all stored in vector<string>&paths. And it will create one output buffer. For each intermediate posting file, we will create an Load object and get the first posting from the Load object, make the first posting and the Load object into a pair structure and put that pair into a priority queue. Then we will do the process same as merging k sorted list to one buffer, when the output buffer is filled, write it into disk in binary format. This function will return a string which is the path of the merged sorted posting binary file.

Algorithm 3 Merge several intermediate posting files into one sorted posting file

Input: vector<string>paths, containing the paths of sorted intermediate posting files, int round, int num, string directory
Output: a string which is the path of merged posting file

function MERGE_TO_ONE_FILE($vector < string > \&paths, round, num, directory$)
  $vector < Load* > load$
  New a Load class for each path and append it to $load$
  Define a pair<Tuple, Load*> type priority queue pq
  for each $v \in load$ **do**
    $tmp \leftarrow v \rightarrow get\_data()$
    push {tmp,v} to pq
  end for
  $OutputFilePath \leftarrow directory +$ "$/sorted$" $+ to\_string(round) +$ "_" $+ to\_string(num)$
  open $OutputFilePath$
  while pq is not empty **do**
    $e \leftarrow pq.top()$
    $q.pop()$
    add $e.fisrt$ to the output buffer
    if the output buffer is filled **then**
      output the content of buffer into $OutputFilePath$
    end if
    $e.first \leftarrow e.second \rightarrow get\_data()$
    if $e.first$ is valid **then**
      push e to pq
    else
      delete $e.second$
    end if
  end while
  return 0
end function

3.4 Build index process

The IndexBuilder class is in build_index_block.cpp.

3.4.1 Constructor of IndexBuilder class

The main function of the constructor is to new memory for input buffer and output buffer. The input buffer is to store the sorted postings, the output buffer is to store the compressed index. It also load the intermediate lexicon from disk since we need to add the start offset and the size of each term's inverted list to the lexicon.

3.4.2 Main function of index building

The build() function is the main function to build index.
It will constantly read a batch of postings from the merged posting file into the input buffer until it have read all postings.
For each term id, it will store the document ids and frequency in two temporary buffer and pass them to reformat_compress() function to encode.

Algorithm 4 Build index

```
1: function BUILD( )
2:     define empty doclist and frelist
3:     LastTermId ← −1
4:     while have not read all postings from disk do
5:         read a batch of postings into the buffer
6:         while buffer not empty do
7:             get a posting from the buffer
8:             if CurrentTermid! = LastTermId then
9:                 build index for LastTermId
10:                clear the doclist and the frelist
11:                LastTermId ← CurrentTermid
12:            end if
13:            append CurrentDcoumentid to doclist
14:            append CurrentFrequency to frelist
15:        end while
16:    end while
17:    return void
18: end function
```

3.4.3 Compute metadata and compress the index list into block

void reformat_compress(int termid,vector<int>&doclist,vector<int>&frelist);
Every time it will read 64 document ids and 64 frequency from doclist and frelist respectively, and encode them to a block by varByte algorithm, record the last document id for each block in a temporary vector called "lastdoc", and record the block size for each block in a temporary vector called "start". To generate a inverted list for a term, it will encode last document ids and then encode block offsets and then append all compressed blocks at the end. Add the start offset and the size of this term's inverted list to the lexicon. Add this inverted list to the output buffer. If the output buffer is filled, write it out into disk.
The varByte compression is shown below:

```
1 vector<unsigned char> varEncode(int num){
2     vector<unsigned char>tmp;
3     while(num>127){
4         tmp.push_back(num&127);
5         num = num >>7;
6     }
7     tmp.push_back(128+num);
8     return tmp;
9 }
```

3.5 Query Process

The InvertedList class is in query.cpp. Each object of this class is used to store one term's inverted list. This class implements openList(), nextGEQ(), getFreq(), closeList() for DAAT method.
And it also implements TAAT() for TAAT method.

1. openList(t): Open and close the inverted list for term t for reading. This function will access the lexicon for term t and get the offset in index file, then read the whole index list into the memory for term t. Then it will decompress metadata and cache the decompressed metadata in memory but keep block compressed in memory.

2. nextGEQ(lp, k): find the next posting in list lp with docID >= k and return its docID. Return value > MAXDID if none exists.

Algorithm 5 nextGEQ

Input: an integer k
Output: The next document id which is larger than or equal to k

```
1: function NEXTGEQ(k)
2:     change ← false
3:     while blockID < blockNum and metadata.lastdocid[blockID] < k do
4:         blockID ← blockID + 1
5:         change ← true
6:     end while
7:     if blockID == blockNum then
8:         return MAX_DOCID+1
9:     end if
10:    if change == true then
11:        free the memory for the last cached block
12:        decompress the blockIDth block and cache it in memory
13:    end if
14:    move the pointer to traverse the document ids in the cached block until find the first one which is
       larger than or equal to k
15:    return the first DocID which is larger than or equal to k
16: end function
```

3. getFreq(k): get the frequency of the current posting in list lp.
   This function share the same pointer in nextGEQ(), so it can return the frequency of without searching.

4. TAAT(unordered_map<int,double>&ms): traverse all document ids and frequency in the inverted list of the term and add the BM25 score to the unordered_map

5. closeList(lp): close the inverted list, free the memory for buffer

6. SnippetGeneration(vector<pair<double,int» &top_result,vector<string>&terms): This function will generate snippet for each document in top 10 results. It will read the text for each document line by line, and match the terms in the current line. It will output the line that has the maximum number of different matched terms.
   For example, suppose the query is "java python C", and there are two lines in a document:

   (a) Python or java? You will think java is better than python after you learn java spring framework.

   (b) My favorite language is C, but sometimes I have to use java and python.

   It will output the second line since the second line has all terms (three) in the query. And the first line just has two terms in the query although the terms' occurrence is much more than the second line.

I will elaborate Conjunctive query with DAAT and Disjunctive query with TAAT methods in the next two sections.

3.6 Conjunctive Query Process

Conjunctive Query is to return the most related documents which contain all of the terms in the query.
I implement the conjunctive query by "Document At A Time" method.

Algorithm 6 Conjunctive Query

Input: vector<string>terms
Output: None

```
1: function CONJUNCTIVE(terms)
2:     num ← terms.size()
3:     for i ← 0 → num − 1 do
4:         invertlist[i].openList(terms[i])
5:     end for
6:     sort invertlist by the number of documents in it
7:     did ← 0
8:     list ← ∅
9:     while did <= MAX_DOCID do did ← invertlist[0].nextGEQ(did)
10:        if did > MAX_DOCID then
11:            break
12:        end ifd ← did
13:        for i ← 1 → num − 1 do
14:            d ← invertlist[i].nextGEQ(did)
15:            if d! = did then
16:                break
17:            end if
18:        end for
19:        if d > did then
20:            did ← d
21:        else
22:            bm25 ← 0
23:            for i ← 0 → num − 1 do
24:                invertlist[i].getFreq(did)
25:                bm25 ← bm25 + BM25(did, terms[i])
26:            end for
27:            append the (bm25, did) to the end of list
28:        end if
29:    end while
30:    for i ← 0 → num − 1 do
31:        invertlist[i].closeList()
32:    end for
33:    get top 10 docments by the BM25 score with a priority queue
34:    generate snippets for the top10 documents
35:    return void
36: end function
```

3.7 Disjunctive Query Process

Disjunctive Query is to return the most related documents which contain at least one of the terms in the query. I implement the disjunctive query by "Term At A Time" method.

Algorithm 7 Disjunctive Query

Input: vector<string>terms
Output: None

```
1: function DISJUNCTIVE(terms)
2:     num ← terms.size()
3:     bm25 ← emptyHashmap < int, double >
4:     for i ← 0 → num − 1 do
5:         invertlist[i].openList(terms[i])
6:         invertlist[i].TAAT(bm25)
7:         invertlist[i].closeList()
8:     end for
9:     get top 10 docments by the BM25 score with a priority queue
10:    generate snippets for the top10 documents
11:    return void
12: end function
```

4 The files' format

4.1 Document table

The document table is saved in doctable.txt.
Each line is a document's information, every line's format is:
Document_id URL NumberOfTerms Offset
Since I assign Document_ids according to the parsed order, the Document_id is in increasing order.
The Offset is the byte offset of <TEXT> from the beginning in trec file since I need to retrieve the text of the page when generating snippet for it.

4.2 Lexicon

The lexicon is saved in final_lexicon.txt
Each line is a term's information, and every line's format is:
term start_offset size number
where size is the length in byte of the inverted list for the term and number denotes how many documents where the word appears.

4.3 Intermediate posting files and Merge posting file

They are all in binary format but without using varbyte compression, so the length of each posting is 12-bytes long in the disk.

4.4 Index file

Suppose there are three blocks For each word, the inverted list's format is:
lastdocid1, lastdocid2, ..., lastdocidn, block_size1, block_size2, ..., block_sizen, block1, block2, ..., blockn
Of course they are all in varbyte binary format.

5 How to run this program

Open a terminal, enter the directory. Firstly, download the dataset
Please use the document collection available on this page: TREC-Deep-Learning-2020
Scroll down the page until you come to a section titled "Document ranking dataset", and download the second link in the table, **msmarco-docs.trec**. Note that this is a collection of 3.2M documents, about 22GB in size.
You can also download it by the command below

Command Line

```
$ wget https://msmarco.blob.core.windows.net/msmarcoranking/msmarco-docs.trec.gz
```

Notice:
The easiest way to run is to use the default configuration.
If you want to use the default configuration to run, jump to 4.5 directly after you complete Step1

5.1 Step1: compile

Command Line

```
$ g++ -O3 parse.cpp  -std=c++11 -o parse
$ g++ -O3 merge.cpp  -std=c++11 -o merge
$ g++ -O3 build_index_block.cpp  -std=c++11 -o build_index_block
```

It will generate three executable files, parse, merge and build_index_block

5.2 Step2: run the parse program

./parse <path1> <path2> <buffer_length>
<path1> is the path of trec file
<path2> is the path of the txt file which will record all the paths of sorted intermediate posting files.
<buffer_length> denotes the maximum number of postings(each posting is 12-bytes) in output buffer, not the maximum number of bytes.
After run this program, you can see it generate many sorted intermediate posting files in ./data directory, and their paths are all recorded in the file whose path is <path2>
For example, after running the command below

Command Line

```
$ ./parse msmarco-docs.trec sortedlist.txt 10000000
```

you can see there's a ./data directory created, and all the paths of intermediate posting files under this directory are recorded in sortedlist.txt (each line is a file's path) . And also, there's a lexicon.txt and a doctable.txt created. They are the lexicon and document table described in the ps2.

5.3 Step3: run the merge program

./merge <path1> <path2> <input_buffer_length> <output_buffer_length>
<path1> is the path of txt file which records all paths of intermediate posting files
<path2> is the path of a txt file which will record the merged posting file, so this file will have only one line
<input_buffer_length> denotes the maximum number of postings in input buffer
<output_buffer_length> denotes the maximum number of postings in output buffer
For example, after running the command below,

```
Command Line
$ ./merge sortedlist.txt mergedresult.txt 100000 1000000
```

it will generate a ./merge_intermediate_result directory where lots of intermediate merged files created, and the final merged posting file's path is stored in mergedresult.txt

5.4 Step4: run the build index program

./build_index_block <path1> <path2> <path3> <input_buffer_length> <output_buffer_length>
<path1> is the path of file which records the path of the final merged posting file
<path2> is the path of the final index file
<path3> is the path of lexicon file generated in Step2 <input_buffer_length> denotes the maximum number of postings in input buffer
<output_buffer_length> denotes the maximum number of bytes in output buffer
For example, after running the command below,

```
Command Line
$ ./build_index_block mergedresult.txt final_index lexicon.txt 100000 1000000
```

it will generate a final_index file and final_lexicon.txt file.

5.5 Use default configuration to parse, merge and build index

If you want to run the program with default configuration, just ignore step2, step3 and step4, and run the command below

```
Command Line
$ ./parse <path of trec file>
$ ./merge
$ ./build_index_block
```

You just need to specify the path of trec file
After you run the three commands,
the index file generated is **final_index**,
the lexicon is stored in **final_lexicon.txt**,
the document table is stored in **doctable.txt**.

5.6 Step5: query process

After you run the program with default configuration in 4.5, you can run the commands below to query

```
Command Line
$ g++ -O3 query.cpp  -std=c++11 -o query
$ ./query <path of trec file>
```

After you run ./query, it will take about one minute to load lexicon and document table into the memory. And after it finish, it will ask you to input your query.
You can append 0 or 1 at the end of your query words to specify conjunctive query or disjunctive query. If you do not append 0 or 1, it will run conjunctive query by default. For example:

Command Line

```
Please input your query words: (input "q" to exit)
python or java, which is better? 0  # this is a conjunctive query
Please input your query words: (input "q" to exit)
python or java, which is better? 1  # this is a disjunctive query
Please input your query words: (input "q" to exit)
python or java, which is better?  # this is a conjunctive query by default
```

After you input your query and hit enter, it will return the top10 documents' ids with their urls, bm25 socres, and also snippet, that is some context of the terms in your query. Note that the terms in the query are all highlighted in the snippet.

6 Test

6.1 Test for index building

Suppose we want to test the word "the" To test the correctness of index files, you can run:

Command Line

```
$ g++ test_parse.cpp -std=c++11 -o test_parse
$ g++ test_index.cpp -std=c++11 -o test_index
$ ./test_parse the <number> msmarco-docs.trec
```

After you run the ./test_parse, it will output how many times the word appears in the first <number> of documents in trec, and output the document ids and their frequency into posting_for_a_word.txt in ASCII format. And it will also output the number of documents it appears in the first <number> of documents in trec in the terminal. Copy that number and paste it in the <number of appearance> field below and run:

Command Line

```
$ ./test_index the <number of appearance>
```

It will read lexicon from disk, seek to the start of the inverted list and read the whole inverted list. Then output the first <number of appearance> document ids and frequencies into posting_index_for_a_word.txt, then use diff command to compare these two files, if there is no output in the terminal after running diff command, they are same, which means the index is built correctly.

Command Line

```
$ diff posting_for_a_word.txt posting_index_for_a_word.txt
```

6.2 Test for query

I do not really write any code to test the query process.
The only thing I did is extracting some representative keywords from a certain document in the dataset, then input query containing these keywords. The document always shows up in the top10 result, and in most cases, the rank is high (top3). That means the query process is correct.

7 Performance

7.1 Performance of index building

7.1.1 Execution Time

Program	Execution Time
Parse	45mins
Merge	2mins
Build index	8mins

The overall execution time is about 55 minutes to 1 hour.

7.1.2 File Size

File	Size
Intermediate posting files	16.0G
Merged posting file	14.3G
Index file	3.0G

7.1.3 Lexicon and Document table size

Data structure	Size
Lexicon	29622373
Document Table	3213835

Notice: You can see the size of merged posting file is different from the overall size of intermediate posting files because I got the file size by using "du -h". "du" doesn't add up the files byte sizes, it measures occupied disk space. When I use "wc -c" command to calculate the byte size for these two files, the results are same. The result of "wc -c" for both files is $15,332,032,044$ bytes. Therefore, they are overall $1,277,669,337$ postings generated.

7.2 Performance of Query

7.2.1 Execution Time

Program	Execution Time
Load Time	1mins
Conjunctive Query Time	0.01s-0.1s
Disjunctive Query Time	0.1s-1.5s

7.2.2 Result Presentation

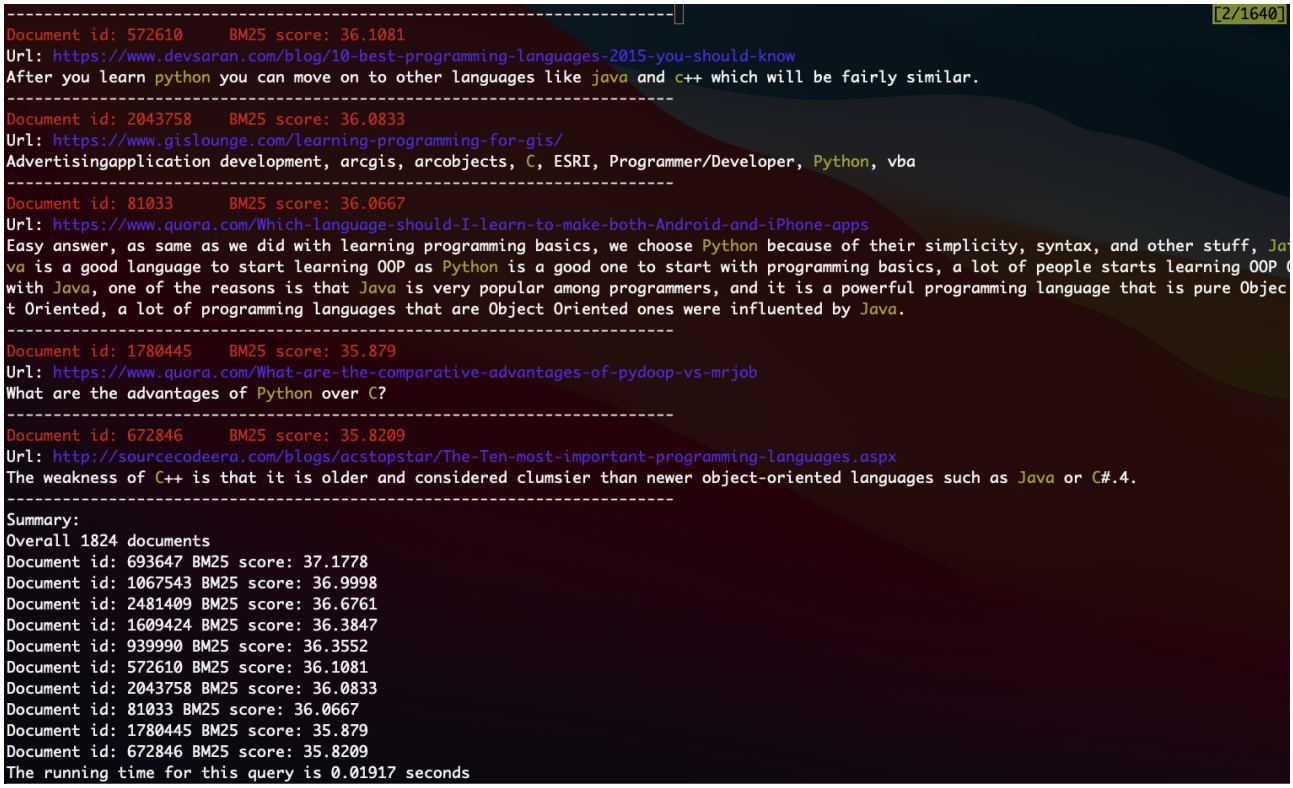

Figure 1: Result of conjunctive query “java python c”

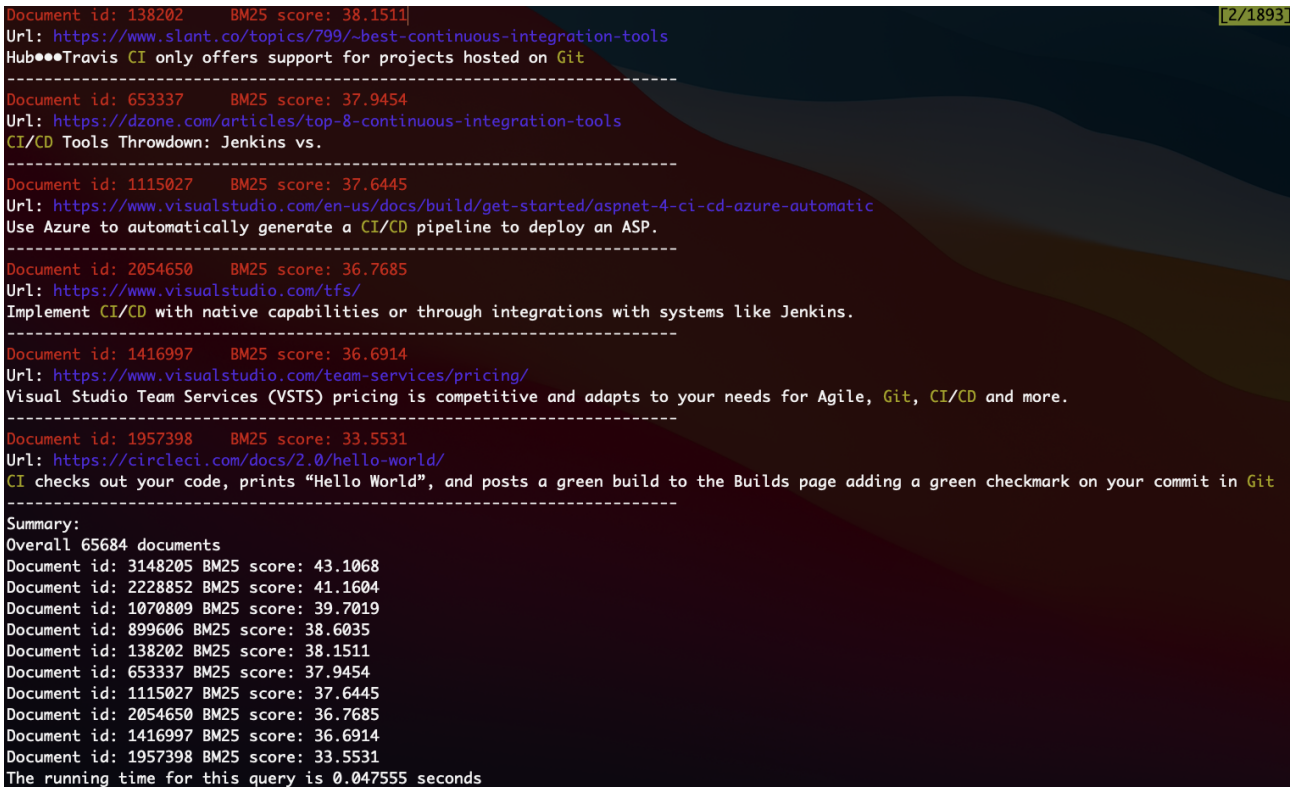

Figure 2: Result of disjunctive query “git CI CD”