CS-GY 6193 **(Due: 11/06/21)**

A simple Web Search Engine

Instructor: Torsten Suel *Name:* Cheng Chen, *Netid:* cc6858

1 Overview

A typical web search engine mainly has three components, namely **crawler**, **indexer** and **query processor**. The crawler fetches a large collection of pages from the web starting from a set of "seed pages". The indexer builds the inverted index structure from the document collection, along with auxiliary structure including page table and lexicon. The query processor answers queries from users. Specifically, it will read inverted lists that correspond to query words, rank the results and generate snippets.

This project makes two simplifications. First, we use an existing document collection from TREC 2020 Deep Learning Track which consists of `3.2M(3,213,835)` pages. Second, the query processor uses a simple rank function BM25, and only answers boolean conjunctive or disjunctive queries.

In summary, this project builds a simple yet complete search engine, which can

(a) Produce a page table, a structure that, given a docID, returns the URL of the page and the size (number of terms) of the page.

(b) Produce a lexicon, a structure that, given a term, returns the start of the corresponding inverted list in the index, the end of the list, and the number of docs containing the term.

(c) Produce the inverted index structure, which stores the actual list for each term. Each list is composed of chunks of 128 docIDs and term frequencies, along with metadata, i.e., arrays with last docIDs and sizes of chunks, at the beginning of the list.

(d) Compute ranked conjunctive and disjunctive queries according to the BM25 measure. Return the top 10 results according to the scoring function. For each top result, return the URL, its BM25 score, and some snippet text.

The project in this paper poses no ethical issues. The rest of the paper is organized as follows: Section 2 introduces the procedure to build and compress the inverted index in detail, as well as the time and space complexity. In Section 3, design decisions of query processor are presented, especially the key operation interfaces. Overall discussion and conclusion are elaborated in Section 4.

2 Indexer

There are generally four algorithms to for index building, namely DIMDS, merge sort, merge subindexes and lexicon partitioning.

I build the inverted index structure based basically on merging subindexes algorithm in four stages indicated by four different source `c++` files.

2.1 Pipeline and Usage

2.1.1 Stage 1: parse.cpp

```
g++ -std=c++11 -O3 parse.cpp && ./a.out 6145
```

- This stage will parse the collection and generate `523` intermediate subindexes named from `index_n1` to `index_n523`. Each subindex is responsible for `3,213,835/523=6,145` documents.
- The inverted lists are arranged alphabetically. The `docIDs` that contain the term are also increasingly sorted. The properties hold throughout all stages.

- This stage will also generate the **page table** structure named `page_table.bin`, assigning `docIDs` in the order the pages are parsed.

2.1.2 Stage 2: phase1merge.cpp

```
g++ -std=c++11 -O3 phase1merge.cpp && ./a.out
```

- This stage will merge the `523` intermediate subindexes into `21` larger intermediate subindexes named from `merge_n1` to `merge_n21`.

2.1.3 Stage 3: phase2merge.cpp

```
g++ -std=c++11 -O3 phase2merge.cpp && ./a.out
```

- This stage will merge the `21` intermediate subindexes into the final inverted index named `merge_nfinal`, which will be converted to compressed binary format in the next stage.
- The total number of indexable terms is `41,085,728`.

2.1.4 Stage 4: compress.cpp

```
g++ -std=c++11 -O3 compress.cpp && ./a.out
```

- This stage will scan the index file and produce the **compressed inverted index** structure named `index_nfinal.bin` in binary format using `var-byte` method, along with the **lexicon** structure named `lexicon`.

2.2 Details

This section explains the key functions/components/design decisions made in building the inverted index.

2.2.1 Stage 1: parse.cpp

- There are generally three different index posting formats for full-text:

```
(docID, frequency)
(docID, impact score)
(docID, freq_1, pos_1, ..., pos_freq)
```

  I choose the first format.

- I use `map<string, vector<uint32_t> > inverted_index` to organize the subindexes. This structure maps a token string to a vector of `docIDs` containing it. When writing to the index file, I use another `map<int, int> df` to parse the vector of `docIDs` into postings, i.e., counting and mapping `docIDs` to its frequences. Subindexes in this phase are lines of

```
term docID,docID,docID,... freq,freq,freq,...
```

- I do not use `termID`.
- The `map` structure of `c++` is internally sorted. The lists are read alphabetically and the `docIDs` are read increasingly from the structure. This structure gets me rid of using Unix `sort`, though might also slow the program down.
- I only consider tokens that only consists of english letter and numbers with the help of `isalnum()`. I also limit the token length to 30 for convenience. However, actually the tokens get filtered by these limitations are not much.

- I use `strtok` to extract tokens from the web page content between `<TEXT>` and `<\TEXT>`. Using a dedicated library might be faster.
- I align the URLs to `1020` bytes which is larger than the maximum URL length. Together with the `4` byte page size, each page table entry consists of `1024` bytes. Therefore the total size for the page table is $1024 \times 3,213,835 = 3,290,967,040$.
- URL byte alignment might help query processing. Given a `docID`, I can multiply it by 1024, seek to this offset on file and read 1024 bytes to get URL and page size. Then we might not need to load the whole page table into memory, though the time to access could be slower. Another disadvantage is the large page table has so many waste null bytes.

```
char page_url[NUM][1020];
uint32_t page_size[NUM];
```

- I accumulate the size of pages and calculate the average length of documents in the collection, i.e., $|d|_{avg}$ to compute BM25 impact score.

2.2.2 Stage 2: phase1merge.cpp

- From stage 1 I will get `523` subindexes, which is not efficient/possible to merge at once. I tried and found that I can only open at most `252` `fstream` at the same time. And it is quite slow to merge this many files. Therefore I decide to merge in 2 passes: `523` $\rightarrow$ `21` $\rightarrow$ `1`.
- I use the `priority queue` structure provided by `c++` to function as min-heap to merge the lists. I construct the inverted list structure as follows.

```
/** structure in the priority queue **/
class elem {
   uint32_t fid;     /** 0,1,2,3, ..., 522 **/
   string token;     /** term **/
   string didlist;   /** docID list **/
   string freqlist; /** frequence list **/
public:
   /** Constructor omitted **/
   uint32_t getID() const {return fid;}
   string  getT() const {return token;}
   string  getDL() const {return didlist;}
   string  getFL() const {return freqlist;}
};
```

- I overload the comparison operator to order the inverted lists, keep them sorted. A list with smaller `token` or `fid` will have higher priority and be popped out first. (This is because the `docIDs` are assigned in the order the pages are parsed, subindexes with larger `fids` will contain larger `docIDs`.)

```
class comp {
public:
    int operator() (const elem* e1, const elem* e2) {
      if (e1->getT() > e2->getT())
         return true;
      else if (e1->getT() == e2->getT())
         return e1->getID() > e2->getID();
      return false;
    }
};
```

- If two inverted lists have the same `token` meaning they should be merged, just simply concatenate their `didlist` and `freqlist` since the first popped list contains smaller `docIDs`.
- The `fid` has another usage, i.e., a new inverted list will be pushed into the `priority queue` from the same `fid` as the popped list until this subindex is exhausted.

- The intermediate subindex files are organized as blocks of three lines:

```
term
docID,docID,docID,...
freq,freq,freq,...
```

2.2.3 Stage 3: phase2merge.cpp

- Same as stage 2 except for the parameters, e.g., file name, file number.
- The final inverted index contains `123,257,184` lines indicating `41,085,728` indexable terms/inverted lists.

2.2.4 Stage 4: compress.cpp

- I implement the `var-byte` method as

```
/** Helper function for Var-Byte encoding **/
vector<uint8_t> buffer;

void VBEncode(uint32_t num){
   while (num > 127) {
      buffer.push_back(num & 127);
      num >>= 7;
   }
   buffer.push_back(num + 128);
}
```

  The buffer is needed to calculate the metadata, e.g., chunk sizes and last `docIDs`.

- I use `128 docIDs` as a chunk. Terms with less than `128 docIDs` will have one chunk.
- During traverse of the `didlist`, I **take differences** of `docIDs`, and then encode those smaller numbers. At the same time, I record the chunk size and last `docID` of each chunk.
- The index layout format is

```
METADATA 128docIDs 128freqs 128docIDs 128freqs ... METADATA ...
```

- I use `uint64_t` to record the start byte offset of the corresponding inverted list in the index and the end of the list in case the `lexicon` size becomes very large. The `lexicon` are lines of

```
term start_of_list end_of_list num_of_docs
```

2.3 Time/Space and Future Direction

This section explains how long it takes on the provided document collection, and how large the resulting index files as well as the intermediate files are. I use `std::chrono library` to record the time.

(a) Stage 1: *parse.cpp*

- `3080.66s`
- `523` intermediate subindexes each around `25M` (text)
- Page table of size `3.1G` (binary)

(b) Stage 2: *phase1merge.cpp*

- `830.20s`
- `21` intermediate subindexes each around `600M` (text)

(c) Stage 3: *phase2merge.cpp*

- `472.60s`
- Uncompressed inverted index of size `12G` (text)

(d) Stage 4: *compress.cpp*

- `665.76s`
- Final compressed inverted index of size `3.2G` (binary)
- Lexicon of size `1.4G` (text)

Regardless of intermediate files, the total size of generated structures is $3.1 + 3.2 + 1.4 = 7.7$G.

The total amount of time used is 5049s $\sim$ 84min, which is not so fast. There are several possible directions to improve the speed.

- Use a more efficient library to parse `.trec` file instead of `getline` and `strtok`.
- Take advantage of `unordered_map` and Unix `sort` instead of `map`.
- Generate less intermediate(temp) subindexes.
- Carefully manage the available/used memory. I just try and error, e.g. `segmentation fault`, throughout the homework.

3 Query Processor

The query processor in this project is rather simple with no intelligence. It does no query rewriting and uses a simple ranking function BM25. It returns 10 results with highest rank score among those satisfying the given Boolean condition.

There are two ways to execute top-k queries using an inverted index, Term-At-A-Time(TAAT) and Document-At-A-Time(DAAT). I choose DAAT as it is generally faster and better than TAAT for search engines.

3.1 Pipeline and Usage

3.1.1 Build and Connect Database

To correctly answer the query, the processor needs to read the complete lexicon and URL table data structures from disk into main memory. The page table contains around `3M` URLs and may need several hundred `MB` memory which is affordable. However, according to my implementation as described in section `2.2.1`, I align the URLs to 1020 bytes so that given a docID, I can multiply it by 1024, seek to this offset on the page table binary file and read 1024 bytes to get URL and page size. Therefore, there is no need to load the whole page table into memory.

The lexicon structure contains around `40M` different terms with maximum length 30, along with `8B` list start offset, `8B` end and `4B` size. The whole structure could use up around `2G` memory which is large. So I decided to use database to store the lexicon and connect it with the processor.

Additionally, to generate the snippet, I need to fetch page content of the top results which should be stored in another database considering that a hash table of page contents is too large to fit in memory. Considering that I need to debug step by step in the building phase, I use **MariaDB python connector** on Jupyter Notebook. The database is set case-insensitive by default and finally the lexicon table contains `36M(36178733)` distinct terms while the documentation table contains `3.2M(3212965)` distinct URLs.

To connect the database and the query processor, I install the **MariaDB cpp connector**, and have included the MariaDB connector header files, specifically `conncpp.hpp`, at the top of the query processor source file.

```
#include <mariadb/conncpp.hpp>
```

3.1.2 Start Searching!

To compile the source file, one has to carefully configure the environment to load the dynamic library.

```
export DYLD_LIBRARY_PATH=/usr/local/lib/mariadb/
```

To start search, simply

```
g++ -std=c++11 -O3 processor.cpp -L /usr/local/lib/mariadb/ -lmariadbcpp && ./a.out
```

And after a greeting message, the processor would ask you whether you want a conjunctive search or not.

```
Conjunctive search?(y/n/quit)
```

Afterwards, you type in several terms separated by space and hit enter, the processor returns the top 10 results according to the BM25 measure. Or, type quit to quit.

A typical search pipeline is shown below.

```
Welcome to Coocle -- a mini web serach engine
          Type 'quit' to quit.
---------------------------------------------
Conjunctive search?(y/n/quit)
>>> y
Please input query...
>>> nyu tandon
Searching...
0: http://www.nyu.edu/search.html?search=cost+of+attendance
   SCORE: 28.4965
      TERM     FREQUENCY
      -------------------
      tandon       1
      nyu          2
      -------------------
   SNIPPET: Search NYU Web (3250) People (0)Cost of Attendance... Attendance . Cost of Attendance
       2018-2019. What's included in the cost of attendance ? ... Cost of Attendance 2018-2019 1.
       Undergraduate
       ...https://www.nyu.edu/admissions/financial-aid-and-scholarships/tuitiongeneral.html65k Cost
       of Attendance... Cost of Attendance 2018-2019. What's included in the cost of attendance ? ...
       Cost of Attendance 2018-2019 1 STERN SCHOOL OF BUSINESS.
       ...https://www.nyu.edu/admissions/financial-aid-and-schola ......

---------------------------------------------
Conjunctive search?(y/n/quit)
>>> quit
Bye~
7.25227s
```

3.2 Details

This section explains the key functions/components/design decisions made in building the query processor.

3.2.1 Build Database

The lexicon, when given a term, returns the start of the corresponding inverted list in the index, the end of the list, and the number of docs containing the term. I create the table as

```
"CREATE TABLE `lexicon` ("
"  `term`  varchar(30) NOT NULL,"
"  `start` bigint     NOT NULL,"
"  `end`   bigint     NOT NULL,"
"  `freq`  int        NOT NULL,"
"  PRIMARY KEY (`term`)"
") ENGINE=InnoDB"
```

As described in section 2.2.4, The layout of `lexicon` structure is

```
term start_of_list end_of_list num_of_docs
term start_of_list end_of_list num_of_docs
...
```

Therefore, just read lines of `lexicon` and

```
term, start, end, freq = line.strip().split()
try:
    cur.execute(
        "INSERT INTO lexicon (term, start, end, freq) VALUES (?, ?, ?, ?)",
        (term, int(start), int(end), int(freq)))
```

The document table, when given a URL, returns the beginning 2048 characters of the content to generate snippet. I create the table as

```
"CREATE TABLE `docs` ("
"  `url`  varchar(1024)  NOT NULL,"
"  `text` varchar(2048)  NOT NULL,"
"  PRIMARY KEY (`url`)"
") ENGINE=InnoDB"
```

Afterwards, traverse the collection *msmarco – docs.trec* and

```
cur.execute(
    "INSERT INTO docs (url, text) VALUES (?, ?)",
    (url, text[:2048])
```

3.2.2 Connect Database

To establish a connection between processor and database, start by retrieving a Driver object that can then be used, in combination with Java Database Connectivity (JDBC) configuration information, to obtain a Connection object.

```
// Instantiate Driver
Driver* driver = sql::mariadb::get_driver_instance();
// Configure Connection
SQLString url("jdbc:mariadb://localhost:3306/");
Properties properties({{"user", "root"}, {"password", "123456"}});
// Establish Connection
Connection* conn(driver->connect(url, properties));
```

To get information from `lexicon` given the query, one has to locate each inverted list for each term by consulting the database.

```
// Create a new PreparedStatement
std::unique_ptr<sql::PreparedStatement> stmnt(conn->prepareStatement("SELECT start, end, freq FROM
    lexicon WHERE term=?"));
/** for each term **/
    // Bind values to SQL statement
    stmnt->setString(1, term);
    // Execute query
    ResultSet *res = stmnt->executeQuery();
    // Loop through and print results
    while (res->next()) {
        start = res->getLong(1);
        end   = res->getLong(2);
        freq  = res->getInt(3);
        /** create list pointer structure **/
    }
}
```

Similarly, we can get the page content for each top URL.

3.2.3 List pointer

Each queried term has an associated data structure called `list pointer`. The structure has interfaces based on operations `nextGEQ()`, `get_score()`, `get_payload()`, etc., on inverted lists. This way, issues such as file input and inverted list compression technique have been completely hidden from the higher-level query processor.

- MetaData

  In addition to the start of the corresponding inverted list in the index, the end of the list, and the number of docs containing the term, each list pointer reads in the metadata, i.e., chunk sizes and last IDs, from the inverted index.

- `var-byte` decoding

  `deque` is a suitable data structure to push back var-bytes and pop from front.

```
/** Helper function for Var-Byte decoding **/
void VBDecode(deque<uint8_t> &dq, vector<uint32_t> &buffer) {
    uint8_t b;
    while (!dq.empty()) {
        uint32_t val = 0, shift = 0;
        b = dq[0];
        while(b < 128) {
            dq.pop_front();
            val += (b << shift);
            shift += 7;
            b = dq[0];
        }
        dq.pop_front();
        val += ((b - 128) << shift);
        buffer.push_back(val);
    }
}
```

- `nextGEQ(k)`

  It is one of the key method of list pointer to implement efficient conjunctive search. It will find the next posting in list with docID $\geq$ k and return its docID. Return -1 if none exists.

  I implement forward seeks in sorted lists block by block.

```
int nextGEQ(int k) {
    while (lastID[cblock] < k) {
        offset   += chukSZ[cblock];
        base_did = lastID[cblock];
        cblock ++;
        if (cblock >= num_of_chks)
            return -1;
    }
    inverted_index.seekg(offset);
    for (int i = 0; i < chukSZ[cblock]; i ++) {
        /** read compressed block **/
    }
    /** uncompress block **/
    cdid = base_did;
    /** traverse block **/
        cdid += buffer[i];
        if (cdid >= k) {
            /** get f_dt **/
            return cdid;
        }
    }
    return -1;
}
```

This implementation doesn't check whether the block is already uncompressed, i.e., it has no cache.

- `get_score`

  BM25 is a ranking function used by search engines to estimate the relevance of documents d to a given search query q.

$$\begin{aligned}\text{BM25}(q,d) &= \sum_{t\in q} s(t,d) \\ &= \sum_{t\in q} \log\left(\frac{N-f_t+0.5}{f_t+0.5}\right) \times \left(\frac{(k_1+1)f_{d,t}}{K+f_{d,t}}\right) \\ K &= k_1 \times \left((1-b)+b\times\frac{|d|}{|d|_{avg}}\right) \\ &= 1.2 \times \left(0.25+0.75\times\frac{|d|}{|d|_{avg}}\right)\end{aligned}$$

  Here, N is the total number of documents in the collection which is 3212965, $|d|_{avg}$ is the average length of documents in the collection which is 1055. These two values are known ahead. f_t is the number of documents that contain term t, which is associated with each inverted list, stored in lexicon. $|d|$ is length of document d, which is associated with each URL, stored in page table. $f_{d,t}$ is the frequency of term t in document d, which is stored in the inverted index.

  In my implementation, the `get_score` method of `list pointer` will calculate and return `s(t, d)` for term t and document d. First, upon constructing `list pointer` given the query, we have f_t from the lexicon. Second, after calling `nextGEQ(k)`, we will get the `docID` in interest along with its $f_{d,t}$. Third, it will seek in the page table to get URL and document size $|d|$. Finally, it has all the parameter values to compute BM25 score and return.

- `get_payload`

  This function is used during a disjunctive search.

  Basically, we have to traverse every posting in the inverted lists for an *OR* semantic search. Therefore I choose to decompress the whole lists, prepare everything ahead and then merge. Here everything including all the `docID`s, URLs, $f_{d,t}$s and $s(t,d)$s.

```
vector<uint32_t> didlist;
vector<uint32_t> freqlist;
vector<double> scorelist;
vector<string> urllist;
```

3.2.4 Search results

Each top result has an associated data structure called `result`, used to construct, sort and display search results. A typical search result format is shown below.

```
0: http://www.audioenglish.org/dictionary/great_wall_of_china.htm
   SCORE: 43.9895
      TERM     FREQUENCY
      -------------------
      china        5
      wall         8
      great        6
      -------------------
   SNIPPET: GREAT WALL OF CHINA Audio English.org Dictionary G Great Bellied ... Great Year GREAT
       WALL OF CHINAPronunciation (US): Dictionary entry overview: What does Great Wall of China
       mean? GREAT WALL OF CHINA (noun) The noun GREAT WALL OF CHINA has 1 sense:1. a fortification
       1,500 miles long built across northern China in the 3rd century BC; is 1,500 miles long and
       averages 6 meters in width Familiarity information: GREAT WALL OF CHINA used as a noun is very
       rare. Dictionary entry details GREA ......
```

This is the top-1 result with index 0 for conjunctive search query "**china great wall**". The impact score, term occurrence and snippet are shown successively.

3.2.5 Conjunctive search

Conjunctive search means AND semantics: "all query words must occur in result". We need to find the `docID`s that occur in all inverted lists in the query first. The impact score can be computed by summing $s(t, d)$ over terms(inverted lists). We then return the `docID`s with top-10 scores by maintaining a top-10 heap.

In my implementation, I use a priority queue to store all the documents that contain all query words and use the impact score as their priority. Then I display search results with the top-10 highest priority.

I use DAAT, i.e., Document-At-A-Time query processing method. DAAT uses no extra space for hash table (only heap for top-10). All inverted lists in the query are traversed simultaneously from left to right. The overall procedure is as follows.

- Sort the inverted lists according to their list length in ascending order.
- Get next posting from shortest list by calling `nextGEQ`.
- Get next posting from other lists and see if you find entries with same `docID`.
- If the `docID` is in intersection, get all the $s(t, d)$s and sum them into impact score. Push the document into the priority queue.
- Increase `docID` to search for next posting.

```
void QueryProcessing() {
    // Intersection
    vector<list_pointer *> lp;
    for (int i = 0; i < num; i ++) lp.push_back(new list_pointer(term, start, end, freq));
    /* sort according to list length(freq) */
    sort(lp.begin(), lp.end());
    int did = 0;
    while (true) {
        // Get next posting from shortest list
        did = lp[0]->nextGEQ(did);
        if (did == -1) break;

        // see if you find entries with same docID in other lists
        int d = -1;
        for (int i = 1; (i < num) && ((d = lp[i]->nextGEQ(did)) == did); i ++) {
        }

        if (d == -1) {
            break;
        } else if (d > did) {
            did = d;
        } else {
            // docID is in intersection; now get all frequencies/scores
            /** details omitted **/
            pq.push(new result(did, url, score, occurrence));
            did ++;
        }
    }
}
```

3.2.6 Disjunctive search

As mentioned in section 3.2.3, we have to traverse every posting in the inverted lists for an OR semantic disjunctive search. Instead of intersection, we have to do a merge of list.

There is a faster disjunctive top-k algorithm called Max Score which makes safe early-termination. Max Score gets the correct top-k without evaluating all docs in union. Max Score is not implemented in this project.

The overall merging procedure is quite similar to that of merging subindexes as described in section 2.2.2.

```
/** structure in the priority queue for merge **/
class elem {
    uint32_t lp;    /** index for list pointer **/
```

```
    uint32_t did;  /** docID **/
    uint32_t freq; /** f_{d,t} **/
    double score;  /** s(d,t) **/
    string url;
public:
   /** Constructor omitted **/
    uint32_t get_lp() const {return lp;}
    uint32_t get_did() const {return did;}
    uint32_t get_freq() const {return freq;}
    double   get_score() const {return score;}
    string   get_url() const {return url;}
};
```

3.2.7 Snippet generation

There are two kinds of snippet, query-independent versus query-dependent snippets. Query-independent snippets are basically summary of page while query-dependent snippets show text surrounding occurrences of the keywords in the page that is useful for user.

Due to the space limit, the database I build for snippet generation only stores the beginning 2048 characters of each page, which probably show no occurrence of query words for a normal-sized page. Therefore, the query processor in this project generate query-independent snippets, beginning 500 characters of each page, for each query. Note that most search engines give query-dependent snippets.

3.3 Future Direction

There are several possible directions to improve the query processor.

- Implement inverted list caching, though the program itself does some caching when seeking in the file.
- Generate query-dependent snippets, probably based on machine learning.
- Leave the decision of conjunctive or disjunctive search to the search engine itself, let it find out which scheme is better suitable for the given query.

4 Discussion and Conclusion

(a) **Performance.** The overall performance of this simple web search engine is pretty good. It provides results in at most one or two seconds for typical queries, especially in the conjunctive case.

We can precompute score at index time and store store quantized term-doc score instead of frequency in the inverted index, to further boost the performance.

```
(docID, impact score)
```

(b) **Quality.** We know that occurrence of "digital" and "camera" doesn't necessary indicate occurrence of "digital camera". Indexing common phrases such as "new york city" or "binary search" can improve the quality of the search results.

By making the database case-sensitive or somehow managing to fit the lexicon into memory can reduce the loss of information.

We can store term occurrence position for smart query-dependent snippet generation.

(c) **Efficiency.** In reality, pure number queries are rare and the search terms are usually much shorter than 30 characters. We can cut down plenty of indexable terms by adding reasonable constraints such as limiting term size to 10 or 15.

We can implement Max Score or other safe early-termination methods to improve the speed of disjunctive search.

We can wrap the STL's priority queue in another class and make it fixed-size to save memory.

This paper understands and builds a simple yet complete web search engine for the document collection from TREC 2020 Deep Learning Track from scratch. We in detail explain the procedure to build the inverted index and query processor, what they can do, how they work internally, what the major functions and modules are, what limitations they have and future directions to improve them, with code snippets. Experiment results confirm that the search engine can provide results in at most one or two seconds for typical queries.