Web Search Engine assignment 3

Kai Chen(kc3443)

Nov. 7, 2018

1 Code structure

- extract_text.py: extract url and text from wet files
- helper.h(helper.cpp): helper for varbyte read and write
- build_index.cpp: build index list
- merge.cpp: merge indexes
- main.cpp: main program to run C++ search engine
- frontend/: angular frontend web page
- frontend/server: node.js server to communicate with C++ search engine

2 How to run

```
### Build

mkdir output
mkdir output/intermediate-output-1
mkdir output/intermediate-output-2
mkdir output/intermediate-output-3

cd extract
# download wet files
python download.py
# extract wet file info to intermediate-output-1
python extract_text.py
cd ..

mkdir build
cd build
cmake ..
make

# build index for files in [0, 100) to intermediate-output-2
# Time : about 5 hours
```

```
./build_index 0 100

# merge [0, 100) indexes to 1 index to intermediate-output-3
# Time : about 45 minutes
./merge 1 0 100 0

cd ..

# add file to communicate with web page
touch in.txt out.txt

### Run C++ Engine
cd build
./main

### Run Web Server (Node Express)
cd frontend/server
npm i
npm run start

### Run Web Frontend (Angular)
cd frontend
npm i
ng serve --open
```

Web Page see Figure1.

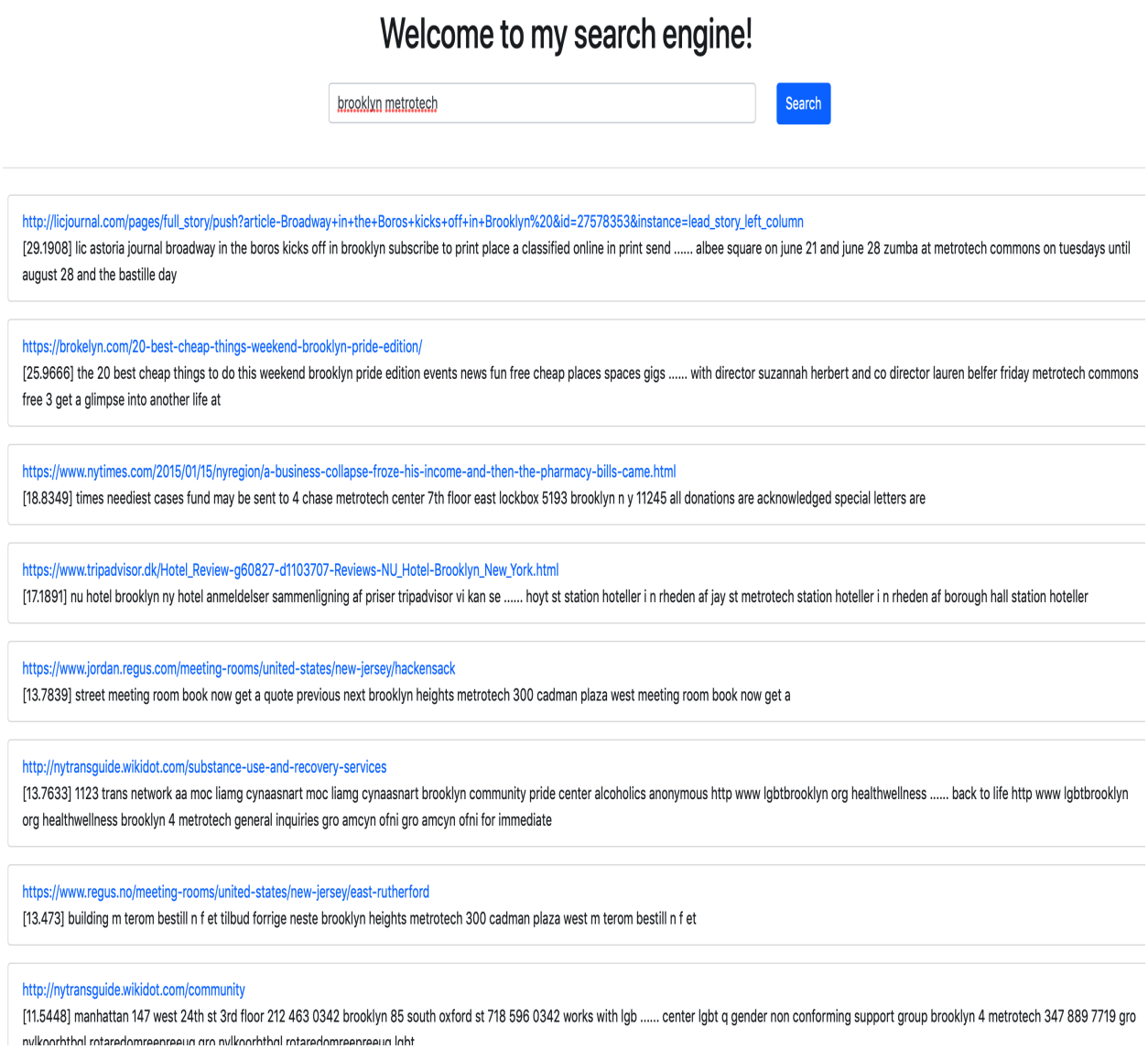

Figure 1: Web Page

3 Build Process

3.1 step1: extract web page

As Figure 2 shows, I use Python to download wet files and extract web information from wet files to txt files. The txt files generated is easy to read for next C++ program.

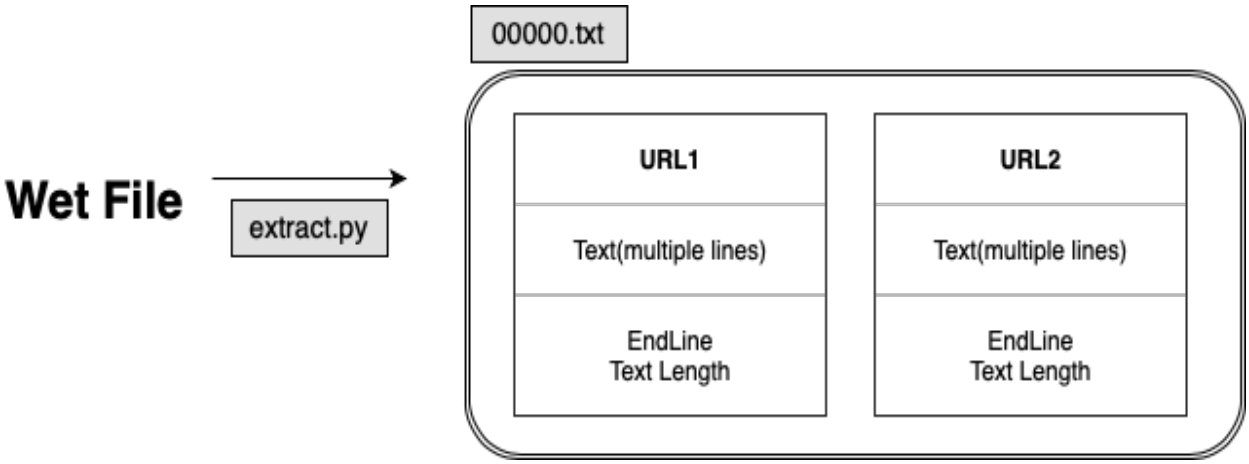

Figure 2: step1 extract web page

3.2 Step2: build index

For all the txt files generated in the above step, I build index files for each one. Index files contains two part, an index table file and an index binary file. Index table file record the offset of terms in the binary file. Index binary file store the list of doc id and frequency for each term(compressed by varbytes).
In the process, I also generate 3 global files: term table file, url table file and url content file. term table record the relationship from term id to term string. url table record the url id to url string and the url content offset in content binary file. url content binary file record content for each url(compressed by varbytes)
The build index process see Figure 3.

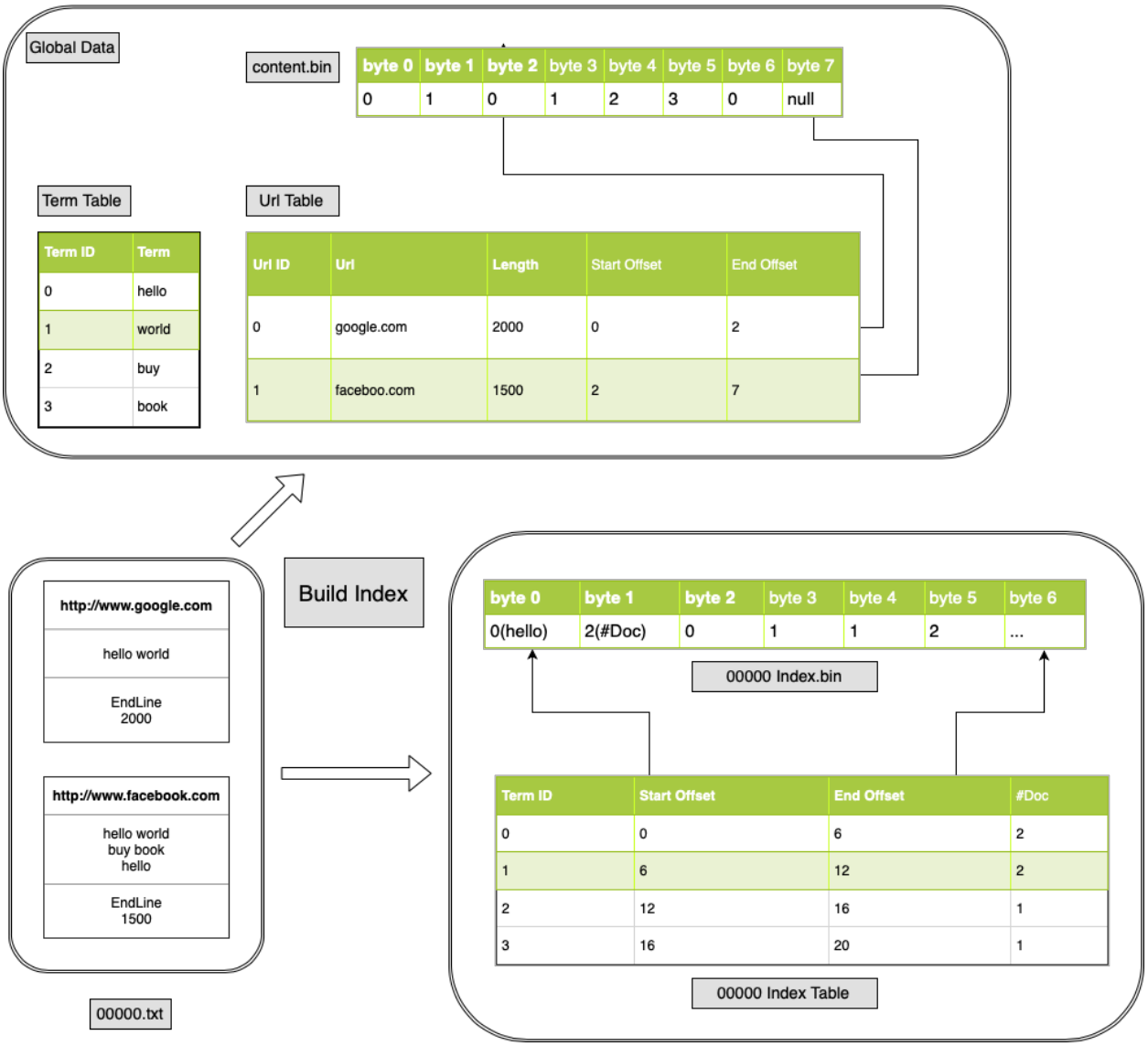

Figure 3: build index

3.3 Step3: merge index

To merge indexes, I use a **min heap** to read the first term of each index. Everytime get the top of the min heap and output the index list to the merged file.
Merge index see Figure 4.

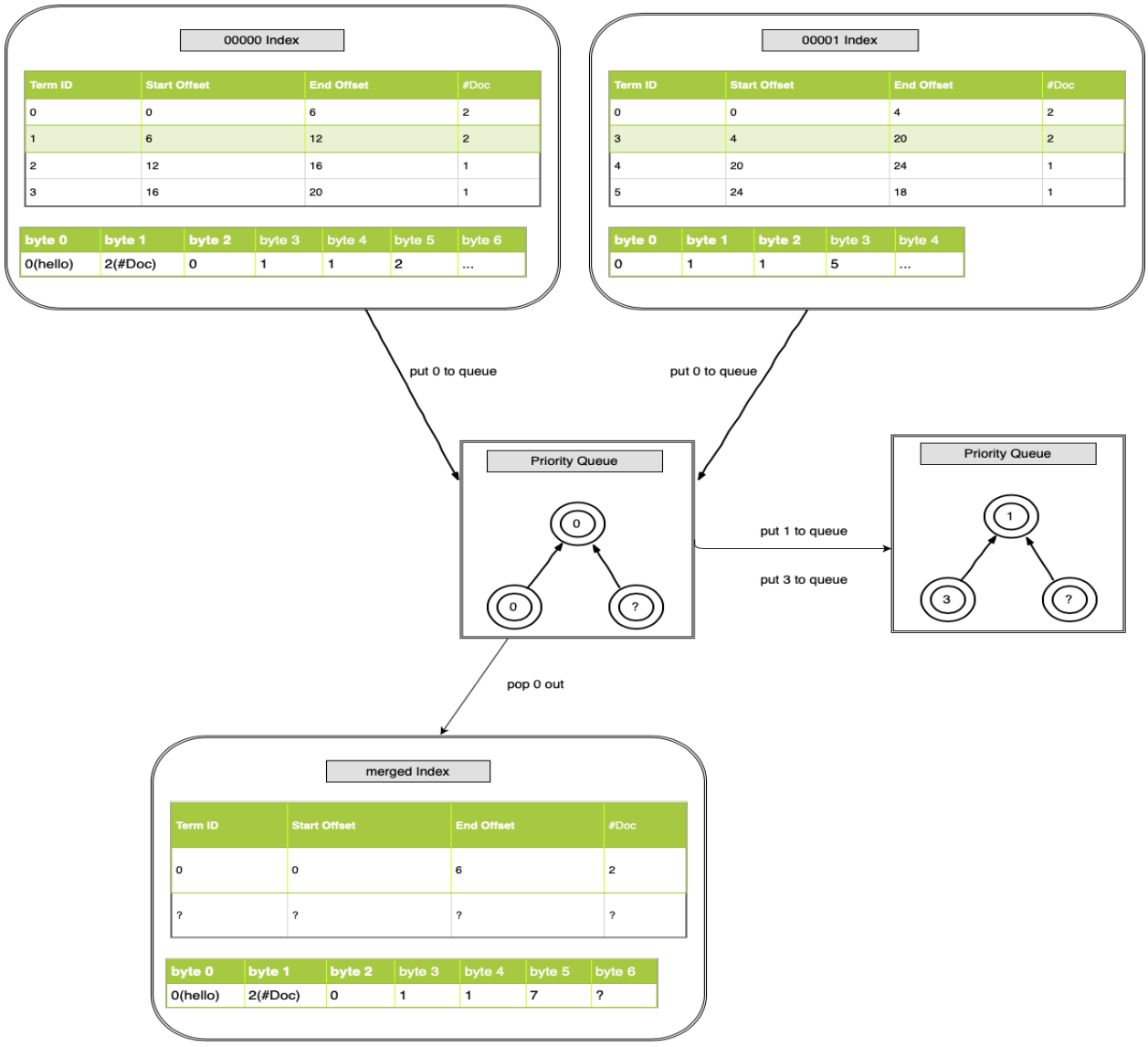

Figure 4: merge index

4 Merge Algorithm

4.1 Algorithm: N-Way merge

To merge 100 intermediate files, I use N-Way Merge. In the **merge_** function, it will call **nway_merge** function below. Firstly, it uses min heap to store the smallest word in each file, every time min heap will pop the smallest item and output it, also remove this term in memory to save space. Then, the next word in file containing the poped word will be pushed to the min heap.

4.2 Code

```
void nway_merge(Reader indexes[], Reader term_indexes[], int n, Writer& merged_index, Wr
    priority_queue<Node> pq;
    map<int, vector<Doc>> mp; // term_id [(doc_id, freq) ..]
    vector<Index> term_indexes_vec[n];
    vector<int> counter(n, 0);
```

```
    for (int i = 0; i < n; i++) {
        term_indexes_vec[i] = term_indexes[i].indexread();
    }
    for (int i = 0; i < n; i++) {
        addToQue(i, pq, mp, indexes[i], term_indexes_vec[i], counter[i]);
    }
    while (pq.size()) {
        auto cur = pq.top();
        int tid = cur.tid;

        // output merged list for term
        output(tid, mp, merged_index, merged_term_index);

        while (pq.size() && tid == pq.top().tid) { // remove all the same term id
            int fid = pq.top().fid;
            pq.pop();
            addToQue(fid, pq, mp, indexes[fid], term_indexes_vec[fid], counter[fid]);
        }

        mp.erase(tid);
    }
}
```

5 Varbyte Implementation

5.1 write int to varbyte

```
void Writer::vwrite(int x) {
    vector<int> digits;
    digits.push_back((x % 128));
    x /= 128;
    while (x) {
        digits.push_back(128 | (x % 128));
        x /= 128;
    }
    int len = (int) digits.size();
    for (int i = len - 1; i >= 0; i--) {
        out << (unsigned char)(digits[i]);
    }
    offset += len;
}
```

5.2 read binary from varbyte

```
vector<int> Reader::vread(long long start, long long end) {
    in.seekg(start);
    long long len = end - start;
    string s;
    s.resize(len);
```

```
    in.read(&s[0], len);
    vector<int> arr;
    for (int i = 0; i < len; ) {
        int val = 0;
        int j = i;
        while (j < len) {
            int ch = (unsigned char)s[j++];
            val = val * 128 + (ch & 127);
            if (ch <= 127) break;
        }
        i = j;
        arr.push_back(val);
    }
    return arr;
}
```

6 DAAT Query With BM25

6.1 Algorithm: DAAT

I use DAAT query. Every time read one document from each query term list, calculate it's BM25 score, and push it to a **min heap**. If the size of min heap is larger than 15(the default top result number), pop the top node of the min heap.
Qeury process see Figure 5.

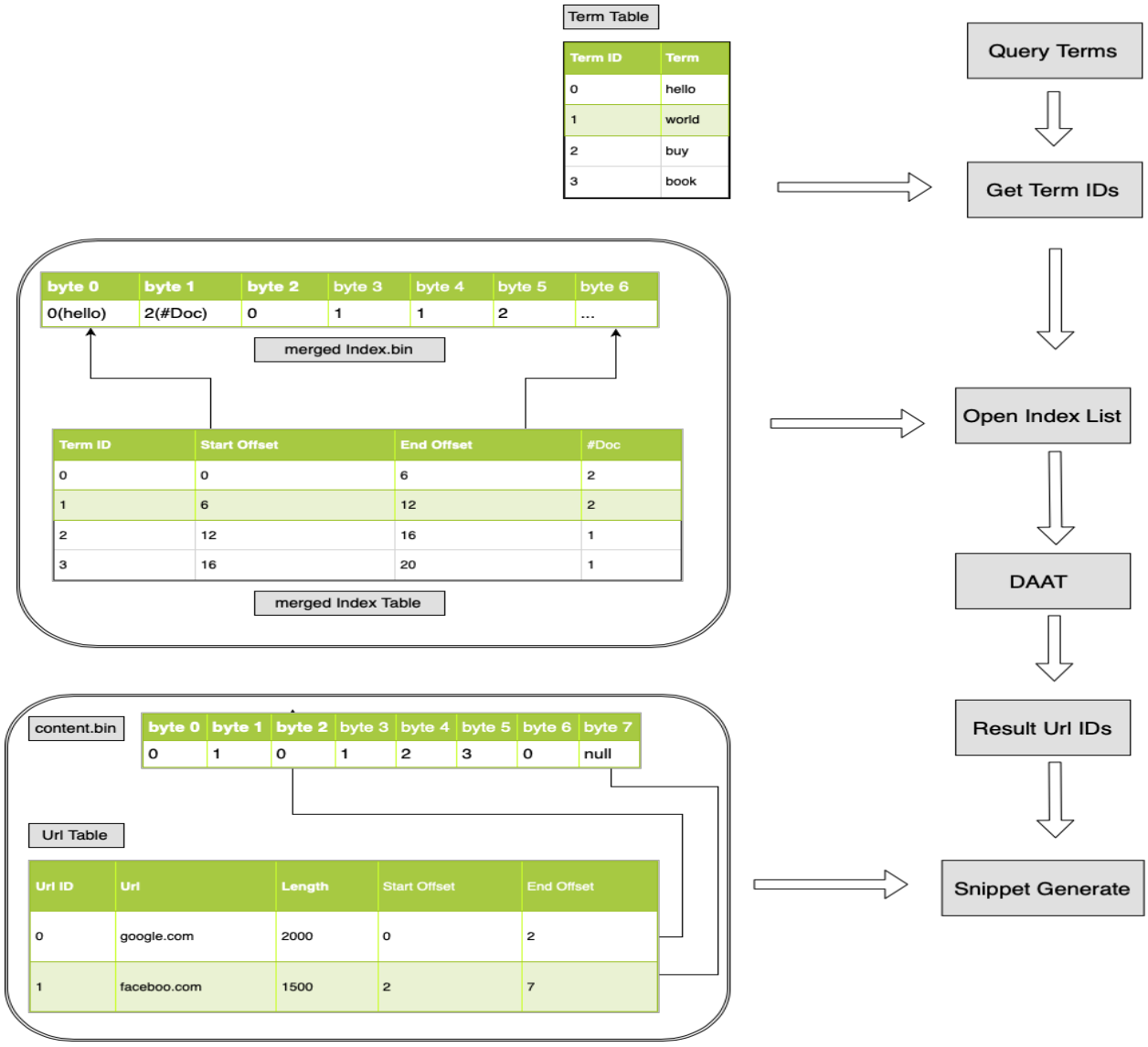

Figure 5: Query Process

6.2 Code

```
vector<pair<int, double>> Query(vector<string> query) {
    if (cache.find(query) != cache.end()) { return cache[query]; }

    int top = 15; // choose top 15
    priority_queue< pair<double, int>, vector<pair<double, int>>, greater<pair<double, i

    vector<int> termIds = ... (get term ids)

    int n = termIds.size();

    vector<StreamReader> readers(n);
    for (int i = 0; i < n; i++) { readers[i] = openList(termIds[i]); }

    int did = 0;
    while (did < MAXDID) {
```

```
        // get next post from shortest list
        did = nextGEQ(readers[0], did);

        int d = did;
        for (int i = 1; (i < n) && ((d = nextGEQ(readers[i], did)) == did); i++);

        if (did == MAXDID) break;
        if (d > did) did = d;
        else {
            vector<int> f_d_t(n, 0);
            vector<int> f_t(n, 0);
            for (int i = 0; i < n; i++) {
                f_d_t[i] = getFreq(readers[i], did);
                f_t[i] = index_table[termIds[i]].number;
            }
            int doc_length = getUrlLen(did);

            // compute BM25
            double score = BM25(doc_length, f_d_t, f_t);

            pq.emplace(score, did);
                if (pq.size() > top) {
                pq.pop();
            }
            did++;
        }
    }

    for (int i = 0; i < n; i++) { closeList(readers[i]); }

    result = ... (get result from priority queue)

    cache[query] = result;
    return result;
}
```

7 Snippet Generation Algorithm

7.1 Algorithm

For each query term, I find it's first occurrence position p in the context, also I define a shift value to be 10, create a segment in range $[p - shift, p + shift]$. Then for each query terms segment, I merge these segment together with $O(nlog(n))$ time complexity. For all the merged segments $[L_1, R_1], [L_2, R_2]...(L_i <= R_i, L_{i+1} > R_i)$, output words in the corresponding position in the document context and connect each segment with a ellipsis '.....'.

7.2 Code

```
string getSnippet(const vector<string>& doc, const vector<string>& query, int shift) {
    vector<pair<int, int>> segments;
    set<string> vis;
    for (auto str : query) {
        vis.insert(str);
    }
    int len = doc.size();
    for (int i = 0; i < len; i++) {
        string str = doc[i];
        if (vis.find(str) != vis.end()) {
            vis.erase(str);
            segments.emplace_back(max(0, i - shift), min(len - 1, i + shift));
        }
    }
    sort(segments.begin(), segments.end());
    int ptr = 1;
    for (int i = 1; i < (int) segments.size(); i++) {
        if (segments[i].first <= segments[ptr - 1].second + 1) {
            segments[ptr - 1].second = max(segments[ptr - 1].second, segments[i].second)
        } else {
            segments[ptr++] = segments[i];
        }
    }
    string snnipet = "";
    const string ellipsis = "..... ";
    for (int i = 0; i < ptr; i++) {
        int l = segments[i].first, r = segments[i].second;
        if (i > 0) snnipet += ellipsis;
        for (int j = l; j <= r; j++) {
            snnipet += doc[j] + " ";
        }
    }
    return snnipet;
}
```

8 Web Page

8.1 Frontend

I use Angular as my frontend framework, which connect with node.js server below.

8.2 Backend

I use Node.js Express as my backend framework. It will connect with the C++ program to get the search result.

8.3 C++ and Node.js Communication

As Figure 6 show, here I use a tricky to achieve this. If user click search button, Node.js will write a query to a txt file which would be shared with C++. Then every 1 seconds, C++ will read query from this txt file and generate results, writing to another txt. Then Node.js can read result from this writen file and show the answer to user.

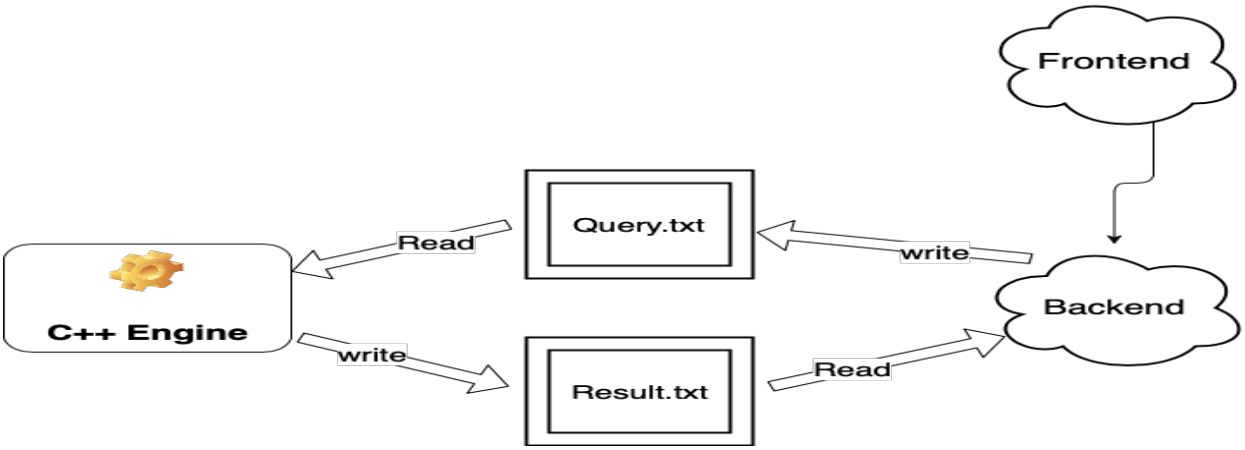

Figure 6: C++ Node.js communication