Information Retrieval Project
Part B, Report

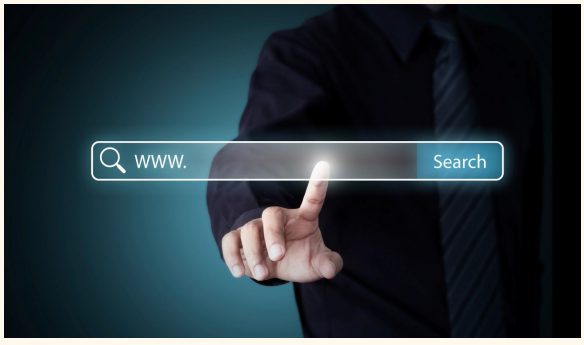
CS172 Project Part B:

Build index and Web-based search interface

Chuliang Zhang - MS in CE, UCR
Email:czhan187@ucr.edu

Jiahao Ge - MS in CS, UCR Email: jge013@ucr.edu

Yingnan Zhang - MS inc CS, UCR
Email: yzhan722@ucr.edu



Abstraction:

Since we want to do a fast information retrieval on our collected data, it is important to build an index with high quality, what's more, in order to improve the efficiency of retrieval, we are supposed to create different fields according to our collected data. Besies, a good rank algorithm is essential for our indexing search. When considering our rank algorithm, since we are dealing with twitter data, the posted time of each twitter record also needs to be considered. In our project, we create an index builder using Lucene with different fields and a web search interface which could connect to our local server which holds created index, data and support lucene search.

TOOLS INSTALLATION PREPARATION

Lucene Libraries: provide indexing and search function

React: javascript library for building userface

uikit : lightweight and modular front-end framework

Node.js: open source server environment used to create our local server

COLLABORATION DETAILS

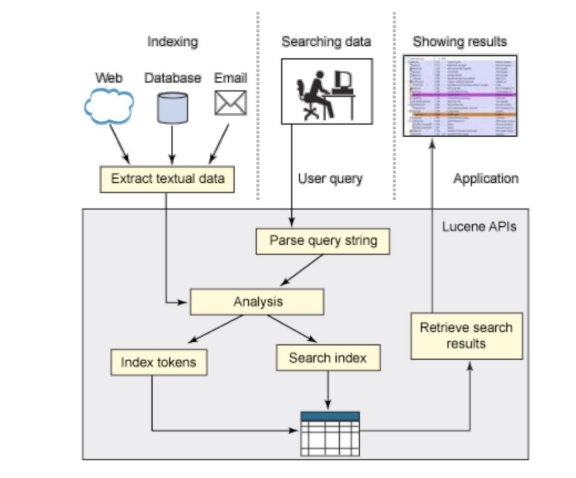
Chuliang Zhang: Twitter Index building and modification and report writing

JiaHao Ge: Web-based interface building, testing and report writing

Yingnan Zhang: System testing and report writing

OVERVIEW OF SYSTEM

1.Architecture



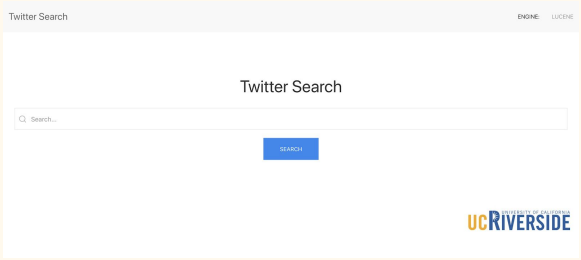
Our system structure looks like the figure above: we have a web interface, lucene index builder engine, lucene searcher engine, local server.

Web Interface:

We use react and uikit to build our web user-interface. Uikit is a lightweight and modular front-end framework, and we modify our user interface based on it.

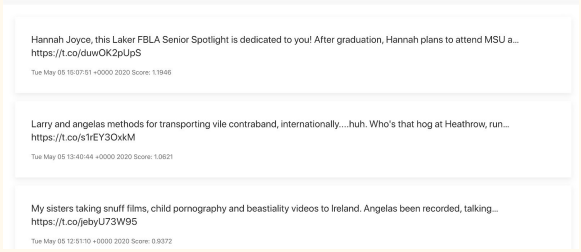
Our user interface looks like figure below:

Search UI:



We have a text input box, the query and key- word typed in here will get and sent to our local server packeted into a request package.

Result presentation UI:



The result of a query searched by our system will be represented like this as a result list of that query, we will show the twitter text content on our result representation page. The result list will be ordered in decreasing order of searching score. And we show the final score of that record and the posted time of that record below the record content.

Lucene index builder engine:

To get the index of files we crawled in Lucene, we need to parse json files first. We use BufferReader to get the data of each line from the json file because each line in the

file is a data with json format, and then enforce it into jsonObject and add it to List<JsonObject>. After parsing files, we need four fields for indexing, which are text, created_at, timestamp_ms and title.

```
BufferedReader bufferReader = new BufferedReader(inputReader);

while ((line = bufferReader.readLine()) != null) {
    // the way of solving the problem of utf-8 with bom
    line = line.replaceAll("\\uFEFF", "");
    obj = (JSONObject) parser.parse(line);
    json.add(obj);
}
```

For text, we use TextField to define it because text is used to search, and we set weight for text since the query will mainly depend on the text field.

For created_at, since we just use it for presenting on the result, so we don't need to set weight.

For datetime, it is used to meet our requirement which the query results will be sorted by the combination of time and score. In order to get normal format, we use DateTools to generate a String format.

For title, we set weight for this field as well and define it as TextField because we use the combination of text and title for query. If text does not have the keyword you search, it will go to title to search.

For each field, it will be added to doc, and then written into a document.

Lucene searcher engine:

For the query, we use httpserver to receive the request from the client. In httpserver, there must be a handler to respond for client requests, and then extract the content of the query from uri and trigger the search function.

```
HttpServerProvider provider = HttpServerProvider.provider();
HttpServer httpserver = provider.createHttpServer(new InetSocketAddress(2019), 200);
httpserver.createContext("/", new MyResponseHandler());
httpserver.setExecutor(Executors.newCachedThreadPool());
httpserver.start();
System.out.println("server started");
```

In the function search, we have multiple fields which can have two or more fields for the query.

Before searching, we need to have a sort to meet with the project requirement that is to use combination of relevance and time to rank.

```
// first sort by score of relevance
// second sort by datetime.
Sort sort = new Sort(new SortField("text", SortField.Type.SCORE),
        new SortField("datetime", SortField.Type.STRING, true));
```

And we just put this sort into search, so we can get the score of the results which is sorted by time and relevance.

After completing the search, we need to transform the data to json format for client so that the data we search could be correctly parsed by web. As for the data from server to client, we just send text, created_at and score to client for display.

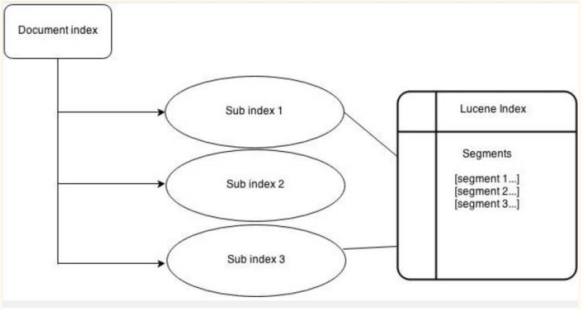
```
// vincent array data for putting searching data as json format
JSONArray array = new JSONArray();
JSONObject data = new JSONObject();

for (ScoreDoc scoreDoc : hits.scoreDocs) {
    Document doc = indexSearch.doc(scoreDoc.doc);
    JSONObject item = new JSONObject();
    // vincent putting data to item corresponding with key like text, title,
    item.put("text", doc.get("text"));
    item.put("created_at", doc.get("created_at"));
    item.put("score", Math.round(scoreDoc.score * 10000.0) / 10000.0);
    array.put(item);
    System.out.println(scoreDoc.score);
}
data.put("data", array);
```

Local server:

We use node.js to implement our server. Our server will get the requests sent from the web-interface and parse the requests then send the query to the lucene search engine, after the lucene search engine gets the result of that query, then the server will return the result to our web interface. We

```
public void handle(HttpExchange exchange) throws IOException {
    // TODO Auto-generated method stub
    System.out.println("receive");

    String requestMethod = exchange.getRequestMethod();

    Headers headers = exchange.getResponseHeaders();
    headers.set("Content-Type", "application/json; charset=utf-8");
    headers.set("Access-Control-Allow-Origin", "*");
    headers.set("Access-Control-Allow-Methods", "GET,POST,PUT,DELETE,OPTIONS");
    headers.set("Access-Control-Allow-Headers", "Origin,X-Requested-With,Content-Type,Accept");

    if (requestMethod.equalsIgnoreCase("GET")) {
        System.out.println("get request");
        Headers responseHeader = exchange.getResponseHeaders();
        responseHeader.set("Content-Type", "text/html;charset=utf-8");

        String response = "this is server";

        Map<String, String> parms = queryToMap(exchange.getRequestURI().getQuery());

        System.out.println("query: " + parms.get("query"));

        String query = parms.get("query");
        String data = null;
        try {
            data = search(indexDir, query);
        } catch (Exception e) {
            // TODO: handle exception
            e.printStackTrace();
        }

        exchange.sendResponseHeaders(HttpURLConnection.HTTP_OK, data.getBytes("UTF-8").length);

        OutputStream responseBody = exchange.getResponseBody();
        OutputStreamWriter writer = new OutputStreamWriter(responseBody, "UTF-8");
        System.out.println(data);
        writer.write(data);
        writer.close();
        responseBody.close();
    }
```

2.Index structure



In our index structure, we support four fields: text, title, created time, timestamp.

Therefore, we could choose which field we want to use and do the query search.

Text field: we store the text content of each twitter record and build our index based on the text field. This field is one of the main components in our search. The size of this field stored will influence the efficiency of the searching system. Since the size of each twitter record is likely to be small, so the weight of text content should be higher, therefore, when we make searching rankings, we put more weight on the text field.

Title filed: in this field, we store the title of each twitter record. This field is also one of the main components in our search. Since the title probably summarizes the general content of each twitter record. However, since the size of the text content of each twitter record will not be too large, we still need to pay more attention to our text

content, not title. So we should not put too high weight on the title.

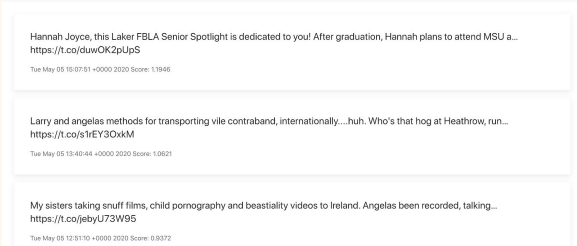
Created_time: since time plays an important role in twitter information search, so we add an additional <created_time> field. We get the <created_time> tag from our json file created on part A. and we create <created_time> field based on it. In our ranking algorithm used on search engines, we combine the created_time with relevance score to get the final score.

Timestamp: After we review the content of some json files created on PartA, we found that not all the twitter records contain the <created_time> tag, so we add a additional <timestamp> field to record the posted time of those twitter records who do not have <created_time>. In this way, we could reduce the number of twitter records without exactly posted time.

3.Search algorithm

In our search algorithm, at first, we will get the score based on the relevancy of query and twitter record. In our project, we use lucene sort.type.score to as our relevance calculation formula. However, we do some modifications on the weight of different fields. We put about 40% on text, and 20% on title, and 40% on created_time, since in twitter, the relevancy and time all are important. And in relevancy, we think if we found content on text is more important, since twitter recore is normally short. So the relevance between query and text content is more important than that between titles.

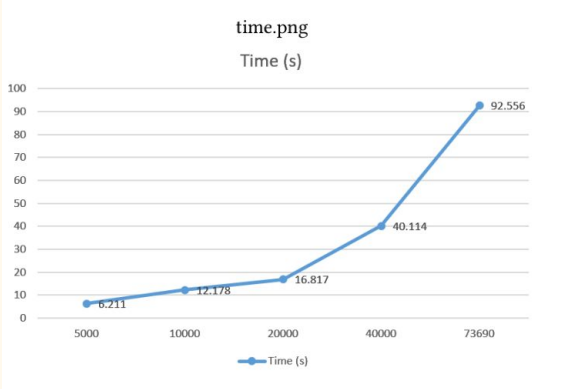
As a result, we will get the final score by adding the relevance score and time score. In our test, we found that if two records have the same relevance score, then the twitter record posted last will be at the front position.



LIMITATION OF SYSTEM

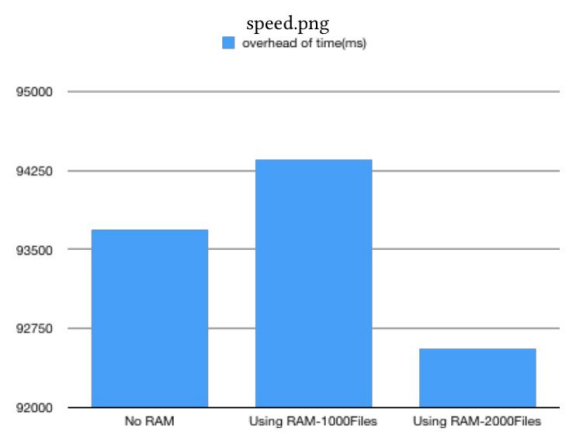
Limitation 1:

Run time of lucene index creation will increase heavily if we have a lot of records that need to be indexed.



To solve this problem, we try to use different RAM write back policies to solve it.

And we found that if we transfer the records from RAM to FileSystem every 2000 files, the overhead of time will be smaller, and speed up our index process.



Limitation 2: Duplication problem

In our system, we did not solve the duplication problem very well. In our test, we found that there will still be some duplication in our result. For the same query, we may get some useless duplicate records which may influence the performance of our system.

Instruction on how to deploy the system

To start the program, you need to ensure you have **node.js** in your system.

First, we just go into the project directory and give the permission for two files called indexsearch.sh and startweb.sh using the command below,

Chmod +x indexsearch.sh startweb.sh

Second, we run the indexsearch.sh like this,

./indexsearch.sh

In this way, you can index the file and start http server.

Third, we run the startweb.sh,

./startweb.sh

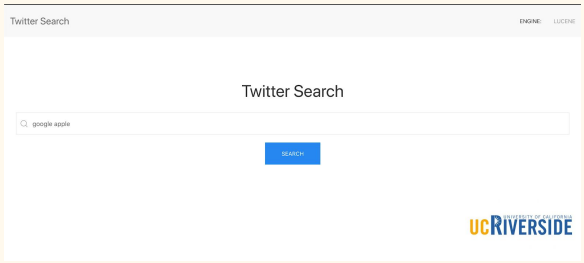
At this point, you start the web client, so you can search any content you want.

In addition, you can use the url below to access the web,

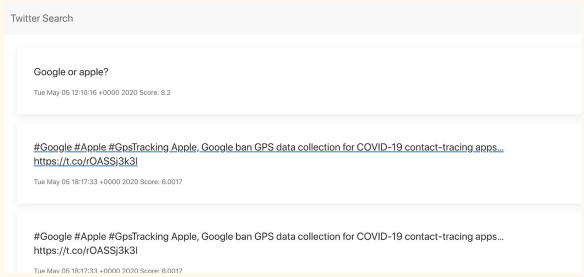
http://localhost:3000/

Screenshots

The screenshot following shows how our system working




```
server started
receive
get request
query: google+apple
Searching google+apple , totalTime is 132ms and the items searched are 1119
8.200039
6.0017185
6.0017185
5.9573183
5.9573183
5.125025
4.756358
4.484048
4.484048
4.2438564
```



Demo video link:

https://drive.google.com/file/d/1eeYFgMytl2v8dVSwz4cvFkNrWjPjjgkq/view?usp=sharing