CS172 - Phase2
Search Engine for Wikipedia

Kevin Nguyen
knguy523

DEMO VIDEO LINK:

https://drive.google.com/file/d/1SS3D1pRBpoQjsakNx8IACII65rR_IjJj/view?usp=sharing

1 INTRODUCTION

This project is about building a search engine for Wikipedia documents collected in phase 1. An index for the documents built using Lucene is used to search documents and return results to display a web frontend. Data collection started on an article about dogs with a depth of 2.

2 TECHNICAL INFORMATION

The backend API uses Java 17 with Spring Boot Web to handle API requests. The web frontend and backend use SvelteKit. Dependencies for the API backend are provided by the maven POM file. The web frontend/backend will need to be installed using Node.js. Installation instructions for Sveletkit can be found in the documentation[1].

Tech Stack	Usage
Lucene==9.2.0	Document indexing and search
spring-boot-starter-data-mongodb==2.6.7	Springboot MongoDB driver
log4j==2.17.2	Logging
spring-boot-starter-web==2.6.7	Stand-alone Spring applications for search API
Svletekit==v1.0.0-next.330	Web frontend/backend

Total index size: **787MB**
Total MongoDB database size: **1.05GB** (MongoDB compression) **1.92GB** (uncompressed)
Total number of documents: **76,375**

2A Architecture

The backend API is a java application run via Spring boot. It is used to index documents stored in MongoDB and then search documents based on user queries. The index is stored on the local file system. The backend API will package the results into a JSON array to serve GET requests.

The web frontend/backend is built with Svelte framework with Tailwind CSS. The routes are handled by SvelteKit. The web app takes in user input and sends get requests to the backend

[1] https://kit.svelte.dev/docs/introduction

API. Responses are then displayed on the frontend with the web page being primarily client-side rendered. Figure 1, below, provides an overview of the system

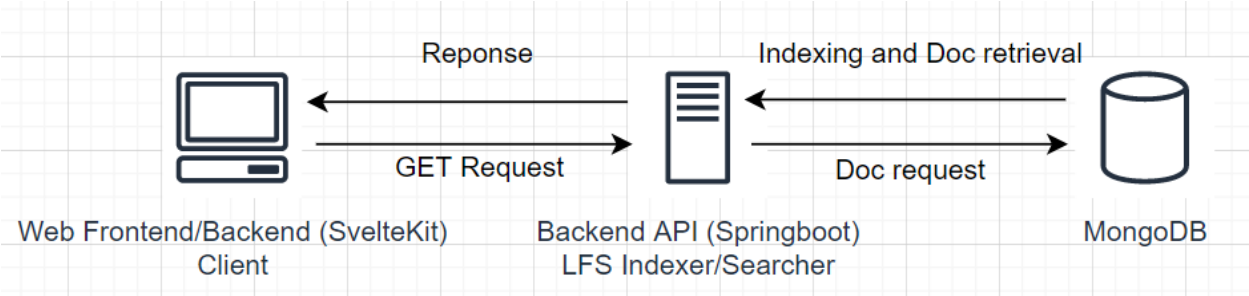

Figure 1: General interaction flow

2B Index Structures

I chose to use Lucene's general field class to get more control over the properties stored in the index. The only property stored in the index is the URL which is used to retrieve documents from MongoDB. The URL is indexed on MongoDB for faster search and retrieval.

Table 1: Index Structures

Field	Properties	Reason
URL	• Stored • Non-analyzed	Useful to store the url but not analyze. Url is indexed on MongoDB for quick retrieval
Title	• Not stored • Index w/ positions and offset • No term vectors	Positions and offsets are for phrase queries
Last Modified	• Not stored • Integer point	Int field for range queries on date modified
Main text	• Not stored • Indexed w/ position and offset • Term vector w/ position and offset	Necessary to store position and offset in index for phrase queries. Necessary to store term vectors for token stream reconstruction and snippet generation
Italic and Bold text	• Not stored • Indexed w/ position and offset • No term vectores	Positions and offsets are for phrase queries. Primary function is for score boosting

Notes about index

The size of the index can be reduced by not storing term vectors and not storing positions and offset of terms. I choose to store term vectors to reconstruct the token stream quickly rather than

generating the token stream again during snippet generation. Positions and offsets are stored for each text field (mainText, italicText, boldText, title) with the index to support phrase queries and proximity queries.

Without specifying fields with the BM25 search option, only the mainText index will be searched. Additional fields can be added with Lucene's advanced search options. If no items are not found in the mainText, the search will be expanded to the title, italic text, and bold text. The last modified field index/stored tree will be used when specified with advanced search queries.

With MultiField BM25 search option mainText, title, and Italic and Bold text will be searched.

```
private static Document indexDoc(org.bson.Document document) {
    Document doc = new Document();
    String lastModified = dateToString(document.getDate( key: "lastMod"), DateTools.Resolution.DAY);
    Field urlField = new Field( name: "url", document.getString( key: "url"), fieldString(Value.store));
    Field titleField = new Field( name: "title", document.getString( key: "title"),
            fieldText(Value.no_store, Value.no_tvector));
    IntPoint dateField = new IntPoint( name: "lastMod", Integer.parseInt(lastModified));
    Field mainTextField = new Field( name: "mainText", document.getString( key: "mainText"),
            fieldText(Value.no_store, Value.pos_offset));
    Field boldTextField = new Field( name: "boldText", document.getString( key: "boldText"),
            fieldText(Value.no_store, Value.no_tvector));
    Field italicTextField = new Field( name: "italicText", document.getString( key: "italicText"),
            fieldText(Value.no_store, Value.no_tvector));
    doc.add(urlField);
    doc.add(titleField);
    doc.add(dateField);
    doc.add(mainTextField);
    doc.add(boldTextField);
    doc.add(italicTextField);
    return doc;
}
```

```
private static FieldType fieldText(Value store, Value tvector) {
    FieldType field;
    // option to store item in index
    field = (store == Value.store) ? new FieldType(TextField.TYPE_STORED) : new FieldType(TextField.TYPE_NOT_STORED);

    // option to store additional index options
    field.setIndexOptions(IndexOptions.DOCS_AND_FREQS_AND_POSITIONS);

    if (tvector == Value.pos_offset) { // option to store term vector
        field.setStoreTermVectors(true);
        field.setStoreTermVectorPositions(true);
        field.setStoreTermVectorOffsets(true);
    }
    return field;
}
```

```
private static FieldType fieldString(Value store) {
    FieldType field;
    field = (store == Value.store) ? new FieldType(TextField.TYPE_STORED) : new FieldType(TextField.TYPE_NOT_STORED);
    field.setIndexOptions(IndexOptions.NONE);
    return field;
}
```

Figure 2: Index options with helper functions to customize indexing and storage settings

The first image of Figure 2 is the insertion of fields. The second image is a helper function to customize the attributes stored with that field. This helper will be used on all types of text fields.

The last image is a helper function that assigns storage options but does not analyze the field. In this case, it is used for the URL.

2C Analyzers

Table 2 provided below summarizes the analyzer used per field. The fields that need to be analyzed are the main text, italic and bold text, and the title.

A custom analyzer is defined with the standard tokenizer with lowercase, stop, English possessive, and Porter stem filters. For my document corpus, stemming often provided better results so I opted for it. Alongside the use stop filter, the stemming lowered the size of the index as well.

The SimpleAnalyzer is used for the title, to keep stop words for more precise searches. This analyzer only uses a letter tokenizer and lowercase filter.

Table 2: Analyzers used on Fields

Field	Analyzer	Reason
Title	SimpleAnalyzer()	Preserves all letter tokenizable words to more exact matches.
Last Modified	Unknown - Point fields are not handled like text	Efficient for numeric range queries (date is processed as integer YYYYMMDD) From lucene docs: “numeric values us(ing) a kd-tree data structure[2]”
Main text	CustomAnalyzer() ● Standard Tokenizer ● Lowercase filter ● Stop words filter ● English possessive filter ● Porter stem filter	Very similar to Lucene’s English analyzer apart from the StandardFilter which further normalizes tokens. English possessive is used to reduce load on the porter stem filter while also getting rid of possessiveness
Italic and Bold text	CustomAnalyzer() ● Standard Tokenizer ● Lowercase filter ● Stop words filter ● English possessive filter ● Porter stem filter	Similar reasoning as above since emphasized text is within the mainText. Emphasized words still benefit from some normalization

The same analyzers used for indexing are used for parsing the query. To help return better results, custom BM25 constants are set based on the fields. Lower k1 values are set for emphasized text and titles because those are relevant fields inherently if words are present,

[2] https://lucene.apache.org/core/9_2_0/core/org/apache/lucene/index/package-summary.html#points

even at lower saturation. Slightly higher b values are used because the length of those fields should matter more.

```
Similarity perFieldSimilarities =  new PerFieldSimilarityWrapper() {
    kevnguy *
    @Override
    public Similarity get(String name) {
        return switch (name) {
            case "title" -> new BM25Similarity(/*k1*/0.5f, /*b*/0.8f);
            case "italicText", "boldText" -> new BM25Similarity(/*k1*/0.6f, /*b*/0.8f);
            default -> new BM25Similarity();
        };
    }
};
```

FIgure 3: Per-field BM25 constants

Figure 3 are the BM25 values used for individual fields. By default, the BM25 has constants b=0.75 and k=1.2.

2D Search Algorithm

There are two options for search:

1. **BM25** - Standard BM25 using Lucene's StandardQueryParser[3]. By default, it will only search the main text of the documents. However, users can build more advanced queries and search fields if they know the syntax and field. This provides the user the most flexibility in choosing which fields to search with, including range queries on the last modified date. When multiple fields are used, BM25 is applied to each field and the score is the summation of all the results. The standard parser is the updated version of the classic libraries.
2. **Multifield BM25** - Standard BM25 with Lucene's classic MultiFieldQueryParser[4]. This provides the user to search through all fields without knowledge of the fields. Custom boosts are applied to emphasize specific fields (e.g title should be more or at least as important as emphasized text). The boost applied to additional fields is set below 1 to keep the mainText score as the dominant weight. The resulting score is the summation of BM25 on the fields. Range queries have little support with the classic parsers due to the construction of the query (each element in the range is built as a boolean query and it quickly hits the maximum query length)

Using advanced queries with option 1 may return similar results as option 2. However, because of the custom boosts on option 2, the ranking of the sites may be different. Because of the use of stemming, the multifield BM25 may also provide more relevant results. This is also due to the analyzer used in the title field. Knowledge of the fields is abstracted from the user with search option 2, making it more intuitive to use. Option 1 provides more advanced features and

[3] https://lucene.apache.org/core/9_2_0/queryparser/org/apache/lucene/queryparser/flexible/standard/StandardQueryParser.html
[4] https://lucene.apache.org/core/9_2_0/queryparser/org/apache/lucene/queryparser/classic/package-summary.html

flexibility with custom queries using Lucene's search syntax, all of which are supported due to the indexing options stored.

```
// Boost values for Multi-field queries
Map<String, Float> boosts = new HashMap<>();
boosts.put("title", 0.15f);
boosts.put("boldText", 0.1f);
boosts.put("italicText", 0.05f);
```

Figure 4: Custom boost on different fields

How BM25 Works in Lucene

Both use the vector space model with a modified BM25 ranking. BM25 is a refinement of the TF*IDF. TF represents the term frequency and IDF calculates how often a word appears in the document collection. Multiplying these terms gives a method of quantifying relevance between queries and documents.

BM25 accounts for term saturation with multi-term queries and document length. This means that documents that match more terms are rewarded but documents that contain repeated words past a saturation point do not contribute as much. This is controlled by the k1 constant. K2 constant is not considered by Lucene and is removed from the standard formula.

To take into account document length BM25 introduces the b constant and the document length divided by the average document length. Shorter documents with high TF are rewarded while longer documents are penalized more.

Surprisingly, Lucene makes little changes to the classic IDF of log(N/DF). Because of the extra 1 in the IDF formula used: log(1 + (N - DF + 0.5) / (DF + 0.5)), the IDF calculation reduces back down to log(N/DF) when the 0.5 terms are ignored[5].

```
2.3227007 = idf, computed as log(1 + (N - n + 0.5) / (n + 0.5)) from:
  7485 = n, number of documents containing term
  76375 = N, total number of documents with field
0.98973554 = tf, computed as freq / (freq + k1 * (1 - b + b * dl / avgdl)) from:
  81.0 = freq, occurrences of term within document
  1.2 = k1, term saturation parameter
  0.75 = b, length normalization parameter
  856.0 = dl, length of field (approximate)
  1426.5518 = avgdl, average length of field
```

Figure 5: Example of Lucene's BM25 calculation

[5] https://kmwllc.com/index.php/2020/03/20/understanding-tf-idf-and-bm-25/

2F Response Objects

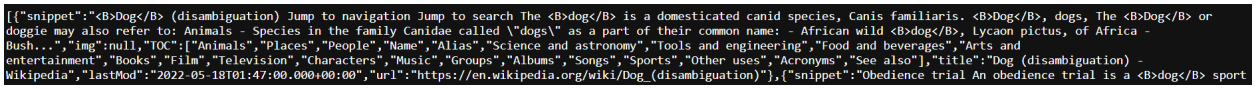

Figure 6: Example response object

Figure 6 shows an example of the object returned to the frontend from the backend API. Fields returned are snippet, title, URL, last modified date, image URL, and table of contents/subheaders. All fields are displayed on the frontend. Only the first ten items in the table of contents are shown due to space and layout constraints. For smaller devices, the table of contents is not shown. If an image was found while scraping, an image icon is displayed in the upper right of the document card. Upon hover, the image will be fetched from the Wikipedia servers and displayed.

```java
public static void main(String[] args) { SpringApplication.run(LuceneRESTController.class, args); }

kevnguy *
@GetMapping(value = "/api/search")
@CrossOrigin(origins = "http://localhost:3000")
public ResponseEntity<ArrayList> HandleGetRequest(
        @RequestParam String q,
        @RequestParam(required = false) String model){
    if(!q.isEmpty()) {
        try{
            ArrayList response = SearchDocs.searchDocuments(q,model);
            return new ResponseEntity<>(response, HttpStatus.OK);
        }
        catch (Exception e){
            throw new ResponseStatusException(HttpStatus.INTERNAL_SERVER_ERROR, e.getMessage());
        }
    }
    else
        throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "Error: Query not provided!");
}
```

Figure 7: Spring Boot GetMapping to handle GET responses

Figure 7 is the code for the single API endpoint. If Lucene can generate a response, the array of documents is returned. Else an error is thrown and a response code is sent to the frontend. Figure 8 showcases the error message on the frontend when Lucene is unable to parse the query.

Figure 8: Error message on the frontend to notify user that search failed

3 LIMITATIONS

The majority of the limitations lie with the analysis for indexing as well as query parsing. Unicode characters are still searchable with the analyzers I used, however; my main focus was a custom analyzer geared towards English text. As a result excess Unicode characters may be stored increasing the index size. Additionally, queries may have to be quite specific due to the inability to normalize Unicode texts properly. There are language-specific analyzers, but I opted not to use them.

The stop filter is used in the analysis of all the text fields beside the title. This means that stop words will only be searchable in the title field. For stemming, I opted to use Lucene's implementation of Porter's algorithm. It is one of the more aggressive stemmers Lucene implements but still less aggressive than Lancaster or snowball. However, in some cases, the Porter stem is more aggressive than desired (e.g animation and animals reduce to anim). Both of these filters have their drawbacks, but in general, I saw better results using them.

The scoring algorithm used is BM25 with some configuration on the constants and boosting of scores. In the end, this does not alleviate any of the drawbacks of BM25 and vector space models. These include the assumption of term independence and the disregard of semantic properties. Additionally, BM25 often biases shorter documents with more query terms. In some cases, these documents may not be useful to the user.

Finally, synonyms filters were not used. This means that only documents that match the words (after stemming) provided in the queries will be returned. If I had more time or someone to help me, a curated synonym list or query expansion for dogs would have been a nice addition.

4 INSTRUCTIONS TO DEPLOY

The backend API is developed with java 17 and built using the spring boot initializer. The API can be run with the command `mvn spring-boot:run`. This starts up the backend on localhost port 8080.

To index documents, you should have an instance of MongoDB running. You can edit the connection setting within the Utilities.java and IndexDocs.java files. Then you can run the command `mvn exec:java -Dexec.mainClass=edu.ucr.cs172.IndexDocs`. The program will write the index files to the directory "index" in the same directory.

The web frontend/backend is not deployed but you can run the app in development mode. You will need Node.js and the following command `npm install && npm run dev --`. This will start the app on localhost port 3000.

5 SAMPLE OUTPUT

```
00:43:18.862 [edu.ucr.cs172.IndexDocs.main()] INFO edu.ucr.cs172.IndexDocs - Indexed document with url: https://en.wikipedia.org/wiki/Electromagnetic_pulse
00:43:18.868 [edu.ucr.cs172.IndexDocs.main()] INFO edu.ucr.cs172.IndexDocs - Indexed document with url: https://en.wikipedia.org/wiki/Nuclear_warfare
00:43:18.868 [edu.ucr.cs172.IndexDocs.main()] INFO edu.ucr.cs172.IndexDocs - Indexed document with url: https://en.wikipedia.org/wiki/Antimatter_weapon
00:43:18.869 [edu.ucr.cs172.IndexDocs.main()] INFO edu.ucr.cs172.IndexDocs - Indexed document with url: https://en.wikipedia.org/wiki/Doomsday_device
00:43:18.870 [edu.ucr.cs172.IndexDocs.main()] INFO edu.ucr.cs172.IndexDocs - Indexed document with url: https://en.wikipedia.org/wiki/Dead_Hand
00:43:18.873 [edu.ucr.cs172.IndexDocs.main()] INFO edu.ucr.cs172.IndexDocs - Indexed document with url: https://en.wikipedia.org/wiki/Mutual_assured_destruction
00:43:18.875 [edu.ucr.cs172.IndexDocs.main()] INFO edu.ucr.cs172.IndexDocs - Indexed document with url: https://en.wikipedia.org/wiki/Relativistic_kill_vehicle
```

Figure 9: Sample output Indexing

FIgure 9 shows sample output when running the indexer. The indexer will create a directory named “index” to store Lucene indexing fragments.

```
Repository interfaces.
2022-05-29 01:11:23.097  INFO 13440 --- [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2022-05-29 01:11:23.104  INFO 13440 --- [  restartedMain] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2022-05-29 01:11:23.105  INFO 13440 --- [  restartedMain] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.62]
2022-05-29 01:11:23.155  INFO 13440 --- [  restartedMain] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2022-05-29 01:11:23.155  INFO 13440 --- [  restartedMain] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 799 ms
2022-05-29 01:11:23.412  INFO 13440 --- [  restartedMain] org.mongodb.driver.cluster               : Cluster created with settings {hosts=[localhost:27017], mode=SINGL
E, requiredClusterType=UNKNOWN, serverSelectionTimeout='30000 ms'}
2022-05-29 01:11:23.464  INFO 13440 --- [localhost:27017] org.mongodb.driver.connection            : Opened connection [connectionId{localValue:1, serverValue:1295}] t
o localhost:27017
2022-05-29 01:11:23.464  INFO 13440 --- [localhost:27017] org.mongodb.driver.connection            : Opened connection [connectionId{localValue:2, serverValue:1296}] t
o localhost:27017
2022-05-29 01:11:23.465  INFO 13440 --- [localhost:27017] org.mongodb.driver.cluster               : Monitor thread successfully connected to server with description S
erverDescription{address=localhost:27017, type=STANDALONE, state=CONNECTED, ok=true, minWireVersion=0, maxWireVersion=13, maxDocumentSize=16777216, logicalSessionTimeo
utMinutes=30, roundTripTimeNanos=18084900}
2022-05-29 01:11:23.575  INFO 13440 --- [  restartedMain] o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on port 35729
2022-05-29 01:11:23.593  INFO 13440 --- [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2022-05-29 01:11:23.600  INFO 13440 --- [  restartedMain] edu.ucr.cs172.LuceneRESTController       : Started LuceneRESTController in 1.557 seconds (JVM running for 1.8
66)
```

Figure 10: Sample output when running spring boot

Figure 10 shows sample output when starting up the backend. If errors are encountered during requests, a message will be displayed here.

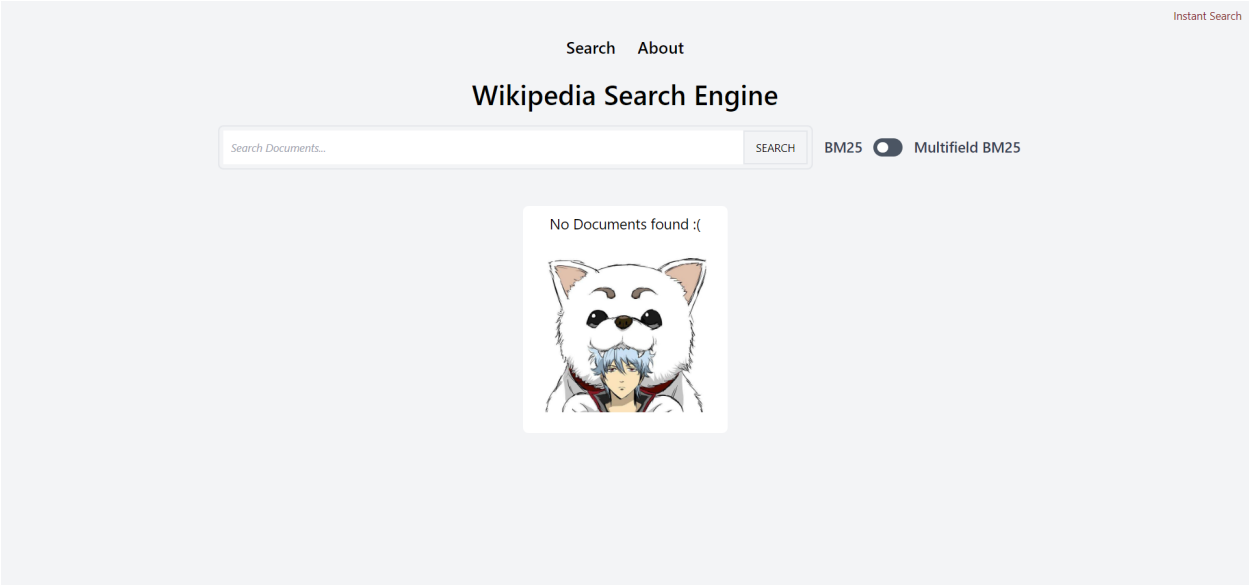

Figure 11: Home Page of web application

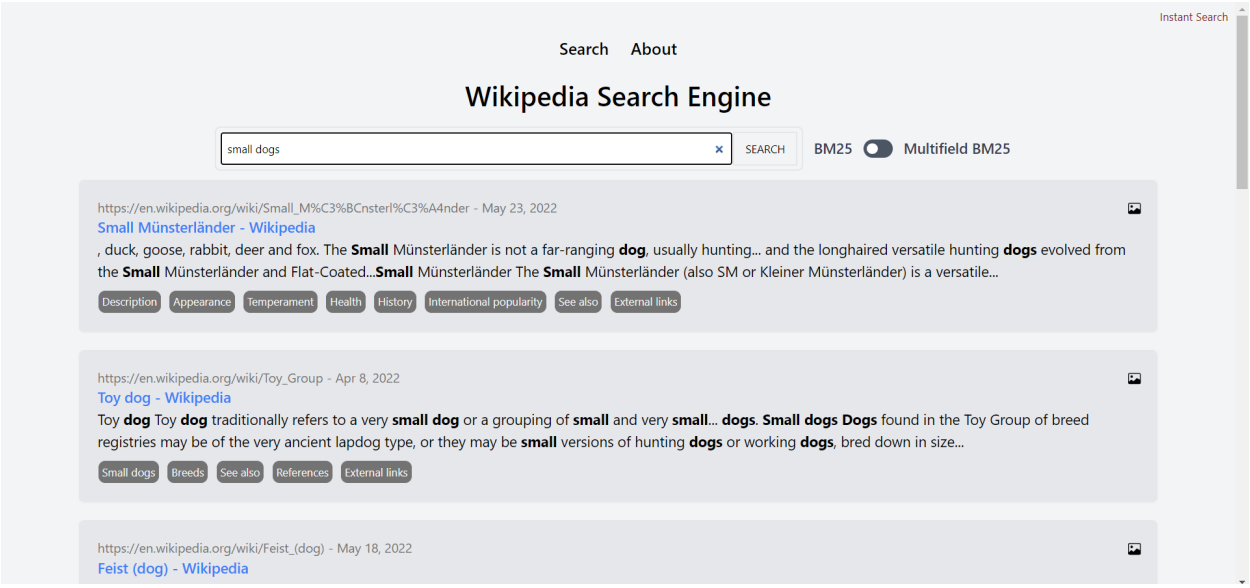

Figure 12: Example search results

Figure 11,12 showcases the frontend in action. Documents are displayed in “document cards” with their fields.

6 EXTRA CREDIT - [NEC = not sure if EC]

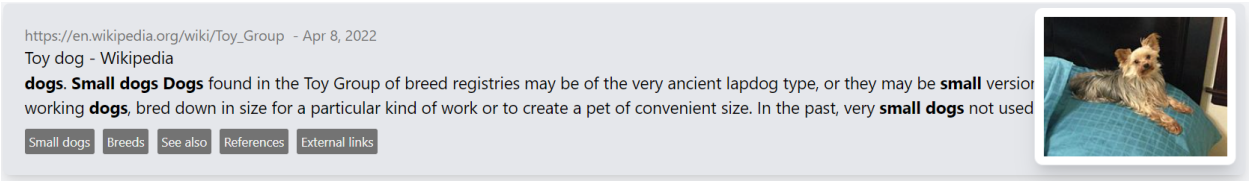

Figure 8: Example search results with image card activated

- Image hovercard if image was found when scraping documents
    - On hovering over the image icon in the top right of document card, the image will be displayed
    - The hover card in Figure 8 disappears when the user moves their mouse from the image/title
- Table of contents scraping and first 10 results displayed
    - User can click on the tag to take them the that section of the wikipedia page
    - TOC are not displayed on smaller devices to save space
- (NEC) Lucene generated snippets
    - Lucene highlighter is used to generate snippets. Scores token fragments and chooses top 3.
- (NEC) Instant search feature - request are sent as user types their query
    - Not very practical