Extended Essay

IB Computer Science

Topic:

Efficiency of inverted index for faster keyword based search on a large volume of unstructured data.

Research Question:

How does the **inverted indexing** technology of **Lucene** compare to that of **MeiliSearch** in terms of **time complexity** for **indexing and searching** on a large volume of textual data?

Essay Word Count: 3836

IBDP Session: May 2022

Table of contents

List of Figures

List of Tables

1. Introduction

The Internet has a vast resource of information today, most of which are unstructured or text-based. When we look for any information on the Internet, one of the most common things we do is google it. So, we search for the information we need based on keywords related to that topic. It's incredible that a search engine like Google can retrieve the data from the Internet's vast resources within seconds.

The volume of data is constantly increasing exponentially, not just because of the increasing number of users but also because of the decrease in price per gigabyte of computational storage. In addition, millions of emails and social media posts are written every minute, which are all unstructured data. Thus, it is even more critical to have a quick and efficient mechanism to find the information out of this pile of data. Modern indexing and searching algorithms make it possible to retrieve data faster and efficiently.

In general, indexing is the process of pointing out the proper location of an object easily and quickly. So, for example, we see indexes being used in libraries to locate books swiftly. Similarly, text files have to be appropriately indexed to search the contents of those files swiftly.

This essay will focus on how effective inverted indexing is for faster keyword-based search on large volumes of unstructured data. This essay will specifically look into luchene and meilisearch, which are two different full-text search libraries. Given a large volume of unstructured data, the time complexity for indexing and retrieving the result of the search for both search libraries will be investigated. Time complexity is a concept in computer science that

deals with the quantification of the amount of time taken by a set of code or algorithms to process or run as a function of the amount of input. (Techopedia Inc., 2018) Hence, the question: How does the inverted indexing technology of Lucene compare to that of MeiliSearch in terms of time complexity for indexing and searching on a large volume of textual data?

2. Background Information and Theory

2.1 Indexing techniques

Data can be broadly classified as structured and unstructured.

Structured data is usually stored in databases, and indexes are generally created based on primary or unique key and corresponding references pointing to the record. Usually, keys are stored in B-tree indexes in databases to navigate to the required search keys quickly.

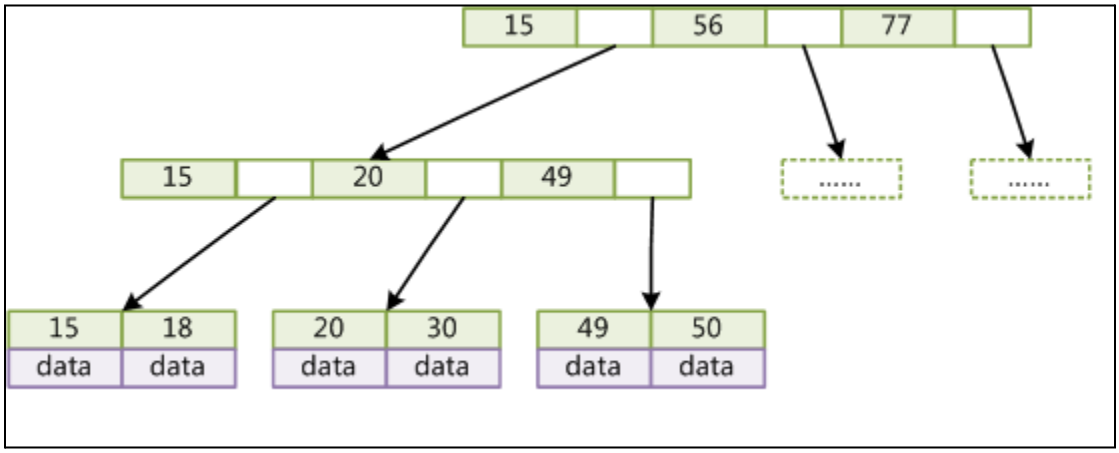

Figure 2.1.1 B Tree Index Example. Image source: (Develop Paper, 2020)

In the case of unstructured data, the above mechanism does not work. Firstly, there may not be specific primary keys as users can search for any words in the text files. Secondly, it is not efficient to store all the text inside a file as a single record because the words must be tokenized (separated) and then indexed.

Now, the question is, how do we index the tokenized words. There are two ways we can index these records: forward index and inverted index.

Forward Index

Forward index is a data structure that stores mappings from documents to words, i.e., directs you from document to word. (Jain, 2018) For example, let's assume there are four documents as following:

Doc1: I like to eat apple orange and banana

Doc2: Apple and orange are my favourite fruit

Doc3: Orange fruit has orange color

Doc4: Fruits have different color

Then the indexing is as follows:

Document	***keywords***
Doc1	i, like, to, eat, apple, orange, and, banana
Doc2	apple, and, orange, are, my, favourite, fruit
Doc3	orange, fruit, has, color
Doc4	fruits, have, different, color

Table 2.1.1 Example of forward index

Note that the same words are stored multiple times in the index. In the above example the word “orange” is stored three times. When a user searches for a particular word the entire index has to

be scanned. Thus, even though it is faster to index it will perform poorly when searching for words, which is not ideal to search upon large volumes of data.

Inverted Index

Inverted index is a data structure that stores mapping from words to documents or set of documents i.e. directs you from word to document. (Jain, 2018) Using the same example of documents in the forward index, the corresponding inverted index will be as following:

Word	***Documents***
and	Doc1, Doc2
apple	Doc1, Doc2
are	Doc2
banana	Doc1
color	Doc3, Doc 4
different	Doc4
eat	Doc1
favourite	Doc2
fruit	Doc2, Doc3
fruits	Doc4

has	Doc3
have	Doc4
i	Doc1
like	Doc1
my	Doc2
orange	Doc1, Doc2, Doc3
to	Doc1

Table 2.1.2 Example of inverted index

As we can see in the example above, the dictionary of words is maintained, and for each word, the index stores in which documents the word is present. The words themselves are sorted alphabetically. Even though there is an overhead in maintaining the word dictionary, it is much faster to search for words as the words are sorted, and we can immediately get the documents having that word. Thus even though it takes more time to index than that of the forward index, it is much quicker to search and retrieve information.

Mechanism to form Inverted Index

Text preprocessing and analysis:

Tokenization

Firstly, the text file that has to be inserted in the index has to be parsed to get all the words in it. Usually, the words are separated by space characters in English, but they can also be other characters in other languages like Chinese. This step is called tokenization.

Stopword Removal

Common words that occur in most of the documents like (a, an, and, are, as, at, be, but, by, for, if, in, into, is, it, no, not, of, on, or, such, that, the, their, then, there, these, they, this, to, was, will, with) (elastic, 2021) are removed. They are removed because these high frequency words are less relevant to search. However, we can opt not to remove these words or add other additional stop words depending on the need.

Stemming

A stem of a word is the word derived by removing the suffix and prefix of an original word. The words having the same meaning in the form of verbs, tenses, and plural forms can be converted to its root word. For example the words go, went, goes, going all can be converted to its root word "go". In English, the most common method to get the stem word is using the port-stemmer algorithm. Indexing algorithms use stemming to get relevant documents when a user searches for any word that is derivative of its root word.

Storing and updating the index:

Term Dictionary

Term dictionary contains all the unique terms extracted from the text preprocessing step. The dictionary itself can have a large number of terms. Thus, it is essential to store them so that it is faster to locate the word in the dictionary and add a word to it when needed. This can be achieved either by using.

Sort based dictionary

In this case, the arrangement is based on storing the tokens in either sorted array or binary trees.

Hash-based dictionary

Hash-based dictionary uses a hash function which computes the hash value of each keyword to its corresponding bucket.

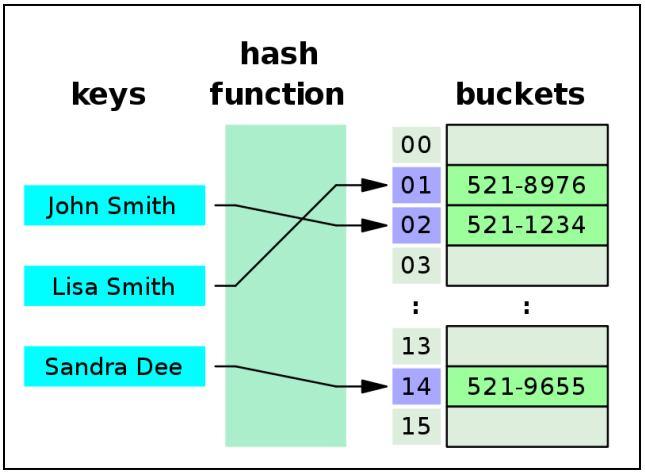

Figure 2.1.2 Hash-based dictionary. Image source: (Stolfi, 2009)

Posting List

The posting list is the sorted skip list of DocIds that contains the related term. It's used to return the documents for the searched term. (Teddie, 2020) It is made up of:

Term - Tokenized word

Document Ordinal - Unique Id of the document

Term Frequency - The number of times the term appears in the document

Term Position array - Positions of the term in the document

Mechanism to search in the index

Parsing query

Different search engines have their own syntax for searching text. The most common functionalities are besides regular word search are:

- **Wildcard search** - e.g., ***eng**** searches for all words starting with eng
- **Boolean operators (AND/OR/NOT)** - e.g., ***apple AND orange*** searches for documents having both apple and orange
- **Grouping** - e.g., ***(apple AND orange) or banana*** searches for documents having both apple and orange or having the word banana
- **Proximity Search** -e.g., ***"apple orange" ~10*** searches for documents having apple and orange within ten words of each other

Retrieving documents matching the query

Next step after parsing the query is to look for the list of documents having each word in the query after expanding wildcards. After obtaining a list of documents for each word, the documents must be aggregated based on a boolean query.

Ranking

The final step is ranking, in which the more relevant documents are sorted towards the top of the list and the least relevant are moved towards the bottom of the list. Though this is optional, it can be a handy feature when a search result has a large number of documents. In most use-cases, users only look for the top few documents (like when we google).

2.2 Lucene

Lucene is a full-text search library in Java which makes it easy to add search functionality to an application or website. (Tan, 2021) The most popular search engines such as Solr and ElasticSearch are built with the core Lucene library.

Indexing in Lucene

Firstly we add Document(s) with Field(s) to IndexWriter, which uses the Analyzer to analyze the Document(s) and then creates/opens/edits indexes as needed and stores/updates them in a Directory. IndexWriter is a tool used for updating and creating indexes. (Tutorials Point, n.d.)

```
// For each file, it creates lucene document having three fields
// Document created from this method can be added to index
private Document getDocument(File file) throws IOException {
    Document document = new Document();

    //index file contents, but do not store the content itself
    Field contentField = new Field(CommonConstants.CONTENTS,
        new FileReader(file));

    //index file name, but do not tokenize
    Field fileNameField = new Field(CommonConstants.FILE_NAME,
        file.getName(),
        Field.Store.YES, Field.Index.NOT_ANALYZED);

    //index file path, but do not tokenize
    Field filePathField = new Field(CommonConstants.FILE_PATH,
        file.getCanonicalPath(),
        Field.Store.YES, Field.Index.NOT_ANALYZED);

    document.add(contentField);
    document.add(fileNameField);
    document.add(filePathField);

    return document;
}
```

Figure 2.2.1 Lucene getDocument() function

The code above creates a Lucene document that has three fields. contentField - this field contains the actual text of the document, which is analyzed and tokenized. fileNameField - contains the name of the file, which is not tokenized. filePathField - contains the full path of the file, which is also not tokenized.

```
// constructor instantiates lucene indexer
public Indexer(String indexDirectoryPath) throws IOException {
    // we are storing the index in file system directory
    // This directory will contain the lucene index files
    Directory indexDirectory
            = FSDirectory.open(Path.of(indexDirectoryPath));

    Analyzer analyzer = new LimitTokenCountAnalyzer(new StandardAnalyzer(), 1000);
    //create the indexer using StandardAnalyzer
    writer = new IndexWriter(indexDirectory,
            new IndexWriterConfig(analyzer));
}
```

Figure 2.2.2 Lucene Indexer() function

The code above creates LuceneIndexWriter, which is configured to store the index in FSdirectory, which is the File System directory. Also, IndexWriter uses StandardAnalyzer, which tokenizes words based on English punctuation characters and stores email address and website address (URL) into a single token.

```
// Get the full path of the file and convert it into lucene document
// then add it into lucene index
private void indexFile(File file) throws IOException {
    //System.out.println("Indexing " + file.getCanonicalPath());
    Document document = getDocument(file);
    writer.addDocument(document);
}
```

Figure 2.2.3 Lucene indexFile() function

Finally, the code above adds each Lucene document for each text file into the index.

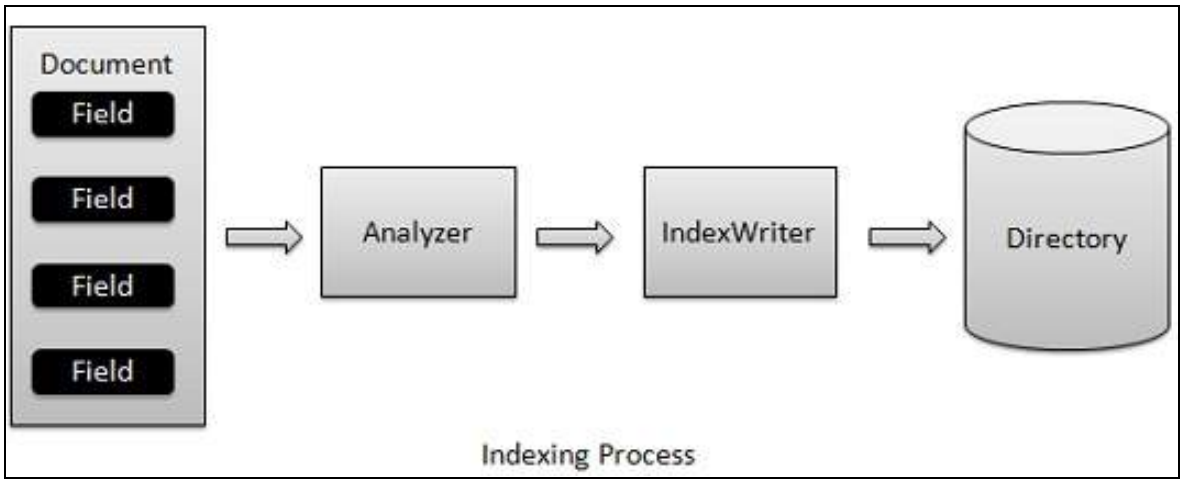

Figure 2.2.4 Indexing process. Image source: (Tutorials Point, n.d.)

The above diagram summarizes the process of indexing, where each document having Lucene fields is processed through analyzer and then written to the directory by IndexWriter.

Internally, Lucene stores the index as a set of files. The following diagram shows some of the essential files that are used for indexing.

*.fnm - stores the information about the fields

*.tis - stores the dictionary of posting list for each term

*.frq - stores which documents have the term and how many times it occurs

*.prx - stores in which position in the document the term occurs

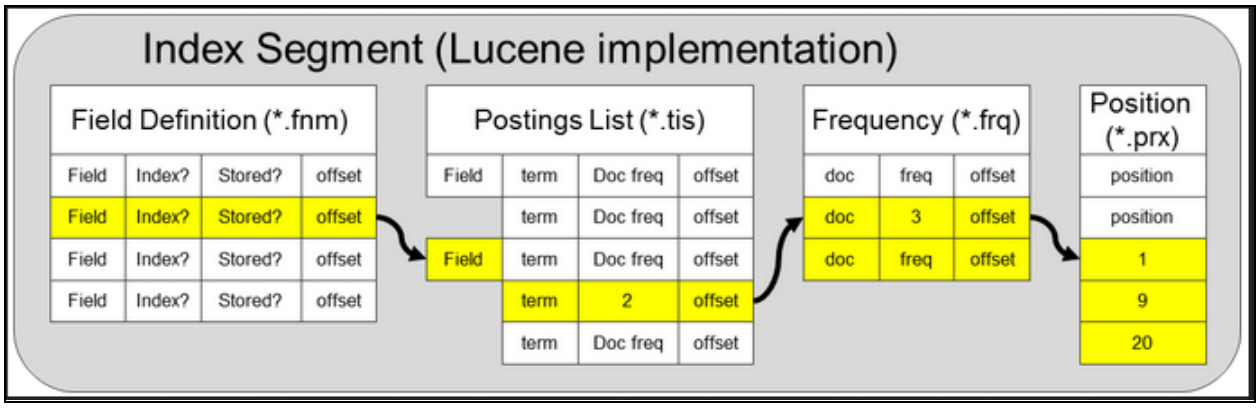

Figure 2.2.5 Index segment. Image source: (Ho, 2013)

Searching in Lucene

For searching, we create a Directory(s) holding indexes for searching and then send it to IndexSearcher, which uses IndexReader to open the directory. Then, by feeding the query to the searcher, we generate a Query with a Term and search using IndexSearcher. IndexSearcher returns a TopDocs object that contains the search details as well as the document ID(s) of the Document that was identified as a consequence of the search. (Tutorials Point, n.d.)

```
// constructor instantiates lucene searcher from index folder
// also instantiates the parser that parses search query
public Searcher(String indexDirectoryPath) throws IOException {

    indexSearcher = createSearcher(indexDirectoryPath);
    queryParser = new QueryParser(CommonConstants.CONTENTS, new StandardAnalyzer());
    queryParser.setAllowLeadingWildcard(true);
}

// Opens the index directory from file system
// and create lucene index reader
// Index searcher uses this index reader
private static IndexSearcher createSearcher(String indexDir) throws IOException {
    Directory dir = FSDirectory.open(Paths.get(indexDir));
    IndexReader reader = DirectoryReader.open(dir);
    IndexSearcher searcher = new IndexSearcher(reader);
    return searcher;
}
```

Figure 2.2.6 Lucene Searcher() and createSearcher() function

QueryParser is created in the constructor using WhiteSpaceAnalyzer so that the same mechanism is used to tokenize search words.

Lucene IndexSearcher is created from the index directory (FSDirectory) stored in the file system by opening it through the IndexReader.

```
// use query parser to parse user query
// then pass the parsed query to lucene searcher
public TopDocs search(String searchQuery)
        throws IOException, ParseException {
    query = queryParser.parse(searchQuery);
    // we can limit the results to show only top N documents
    // for our purpose, we are returning all documents using MAX_SEARCH
    return indexSearcher.search(query, CommonConstants.MAX_SEARCH);
}
```

Figure 2.2.7 Lucene search() function

QueryParser parses the search query to identify the required terms to be searched and identifies wildcards, boolean operators if any. It will form the Query object, which will be used by indexSearcher to get the hits in the TopDocs object. TopDocs contains the list of ScoredDoc, which is nothing but a document with a relevancy score.

```
//get document in search result based on ranking
public Document getDocument(ScoreDoc scoreDoc)
        throws CorruptIndexException, IOException {
    return indexSearcher.doc(scoreDoc.doc);
}
```

Figure 2.2.8 Lucene getDocument() function

Using ScoredDoc, we can fetch the Lucene Document object. With this object, we can display any fields (e.g., filename of the path).

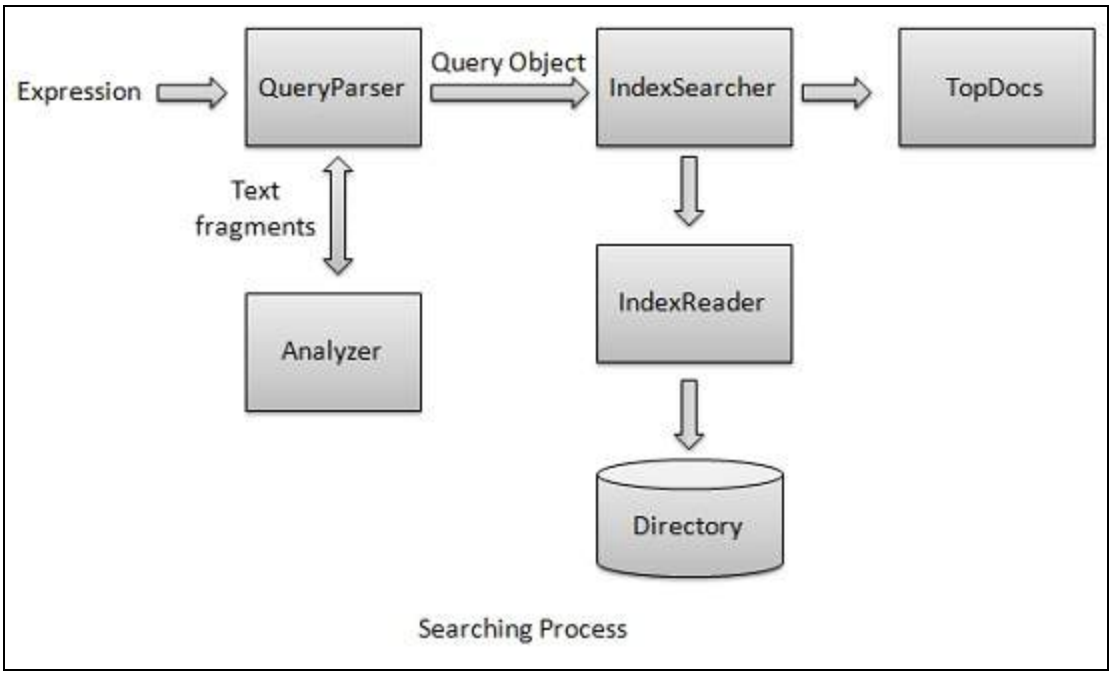

Figure 2.2.9 Searching Process. Image source: (Tutorials Point, n.d.)

The above document shows the overall operation carried out from the code segments above.

2.3 MeiliSearch

MeiliSearch is a next-generation open-source search engine that provides functionality to search within millions of documents getting the most relevant results within milliseconds.

MeiliSearch is based on REST APIs and requires the API server to be running when indexing and searching.

Indexing in MeiliSearch

The storage engine of MeiliSearch is a Lightning Memory-Mapped Database (LMDB for short). When documents are added to a MeiliSearch index, an abstract interface called an analyzer handles the tokenization process. The analyzer is responsible for determining the primary

language of each field based on the scripts (e.g., Latin alphabet, Chinese hanzi, etc.) that are present there. Then, it applies the corresponding pipeline to each field. (Meili Search, 2021)

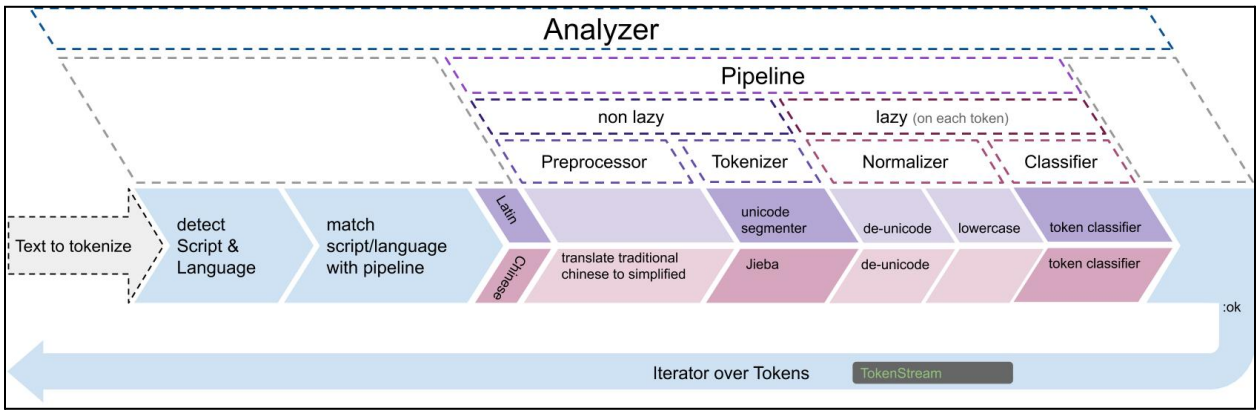

Figure 2.3.1 Indexing process. Image source: (Meili Search, 2021)

```
// create meilisearch client which communicates with meilisearch REST server
Client client = new Client(new Config("http://localhost:7700", "masterKey"));

// Create index if it is not already created
Index index;
try {
    index = client.getIndex(INDEX_NAME);
} catch (MeiliSearchApiException mex) {
    // If index is not yet created, create it and set fields to be displayed
    // An index is where the documents are stored
    index = client.createIndex(INDEX_NAME);
}
```

Figure 2.3.2 MeiliSearch indexing initializing client source code

Before using MeiliSearch, we need to run the server which listens by default on port 7700. Though it provides REST based GET and POST APIs, we are not directly using this API. Instead we activate it through java API which calls the REST apis internally using Client class.

```
JSONArray docList = new JSONArray();

// Applies file filter and fetches all files in directory
for (File file : listOfFiles) {
    if (!file.isDirectory()
            && !file.isHidden()
            && file.exists()
            && file.canRead()
            && filter.accept(file)) {
        String content = readFileContent(file.getPath());

        JSONObject obj = new JSONObject();
        fileId += 1;
        obj.put(CommonConstants.FILE_ID, fileId);
        obj.put(CommonConstants.FILE_NAME, file.getName());
        obj.put(CommonConstants.CONTENTS, content);

        // create a batch of 1000 files at a time and store them in docList
        if (fileId % 5000 != 0) {
            docList.put(obj);
            System.out.print("\33[2K\r Doc Id: " + fileId);
        } else {
            // once the batch of 1000 is reached, then index those 1000 files
            IndexJson(docList.toString(), index);
            docList = new JSONArray();
        }
    }
}
// if there are files in docList that have not reached  batch of 1000 yet
// index those reamining files in the batch
if (docList.length() > 0) {
    IndexJson(docList.toString(), index);
}
```

Figure 2.3.3 MeiliSearch file iteration and index queue function

Since the java APIs internally use REST APIs, we need to store the document to be indexed in JSON format. For each file, we create a JSONObject containing three fields - file id, file name, and contents of the file. DocList contains an array of such JSONObject. As soon as the array count reaches 5000, we queue it to be written in the index. This is done as it will be faster than queueing it one by one.

Searching in MeiliSearch

The query format of MeiliSearch is simple and geared towards getting the most relevant results rather than exact matches. It applies concatenation and splitting on the query to guess the important words in it.

When a search query with numerous words is concatenated, the terms that follow each other are also concatenated. (Meili Search, 2021) For example, query “news paper” is also combined as newspaper so it will search for either news, paper or newspaper. It also applies a splitting algorithm to each word based on the frequency of separate words in the dictionary. If both split parts have significant results, then it will include the results having both words together but separated by spaces. For example, the word newspaper is also split as news and paper.

```
Client client = new Client(new Config("http://localhost:7700", "masterKey"));

// instantite index and configure fields to be displayed
Index index = client.index("stories");
String[] displayedFields = {CommonConstants.FILE_ID, CommonConstants.FILE_NAME};
index.getSettings().setDisplayedAttributes(displayedFields);

// input search query from console
String searchQuery = new InputOutputManager().InputString("Search Query:");

long startTime = System.currentTimeMillis();

// Fetch search results
SearchRequest searchRequest = new SearchRequest();
// set limit to max so that all results are returned
searchRequest.setLimit(CommonConstants.MAX_SEARCH);
searchRequest.setQ(searchQuery);

SearchResult results = index.search(searchRequest);

long endTime = System.currentTimeMillis();

// resutls are in json array
JSONObject obj = new JSONObject(results);
JSONArray hits = obj.getJSONArray("hits");
```

Figure 2.3.4 MeiliSearch searching source code

Again, we need to instantiate the Client object which communicates with the search server. We set the search query and the limit of the number of hits in the SearchRequest object and pass it to the Index object to search within it. The result is also obtained in JSONArray, which we can display.

3. Hypothesis and Applied Theory

The theory of both Lucene and MeiliSearch and how they work has been explained in detail. Now we have to decide which of the two is more efficient in indexing and which one is more efficient in searching. The efficiency will be measured considering the time complexity. Experiments will be carried out to measure the time taken to index and search in the index for both Lucene and Meilisearch. For this, we can use a dataset having a different number of text files to see if one is better in a smaller set and the other is better in a larger set.

Though both creates and stores the parsed text in an inverted index, Lucene stores the index in the file system, while MeiliSearch stores it in a key-value-based database. We expect MeiliSearch to be slower to index because it needs to update the database every time we add documents in the index, while Lucene uses a filesystem and also utilizes system memory to update the index.

We expect the average time taken to search the specific keywords to be similar because it is quick to retrieve information from databases in the case of MeiliSearch as well as from structured index files in the case of Lucene once the information is organized properly as an inverted index.

As for the results of the search, we expect some differences. This is because MeiliSearch automatically expands the search term to include words containing the searched word (.e.g. Searching for the word ‘go’, will also include the word ‘going’). It is also typo tolerant as it includes words with one character spelling mistakes. Typo is an error (as of spelling) in typed or

typeset material. (Merriam-Webster, n.d.) When providing multiple search terms, MeiliSearch also searches for concatenated combinations, while Lucene searches only for exact matches. For Lucene, we are using WhiteSpaceAnalyzer which tokenize words based on whitespace characters. Search results will only include exactly matching words unless we specify wildcards to expand the search-words.

4. Methodology

4.1 Independent variable

The independent variable here refers to the data-set having text files that will be used while conducting the experiment. We will use different data sets having different numbers of files in them. We'll also conduct searches using a variety of terms and keyword combinations, all of which are independent variables.

4.2 Dependent variable

The time it takes to generate an index of the provided data and the time it takes to search a specific word from the index will be the sole dependent variables examined in this experiment. This will be measured after indexing the data set and the amount of time required to search specific words and combinations of them. The time will be measured in milliseconds.

4.3 Controlled variable

Variable	***Description***	***Specifications (if applicable)***
Computer and operating system used	I will be running the program on my laptop: Lenovo Thinkpad X230 with a Linux operating system.	**Distro**: Manjaro Linux **Kernel**: 5.10.42-1-MANJARO x86_64 **Desktop**: Xfce 4.16.0 **Processor**: Intel Core i5-3320M

		Memory: 8GB DDR3
Integrated development environment (IDE) used	I will be using a single IDE to develop my program. JDK platform 15 is used for MeiliSearch while platform 16 is used for Lucene. This is because of the incompatibility of dependent libraries.	**IDE**: Apache NetBeans IDE 12.0 **Java Development Kit**: JDK platform 15, JDK platform 16
Same data used	Set of text files containing (1000, 5000 , 10,000 and 50,000 files)	Each set contains a folder of text files in the English language
Deployment	The program will be launched on the Linux terminal.	Xfce 4-terminal 0.8.10

4.4 Procedure

The procedure of the experiment is as follows:

1. Download sample text files and create data-sets of 1000, 5000, 10000, and 50000 files from the sample downloaded.
2. Set up the environment and the program to index and search the files from both Lucene and MeiliSearch.
3. Run the program to make an index from both Lucene and MeiliSearch and have it return the time taken.

4. Run the program to search the index with specific words and have it return the time taken to search the specific words. For our purpose, we have used the following keywords to search - people, happy people, very happy people, very happy sad people
5. Repeat steps 3 and 4 for each data set.

5. Results

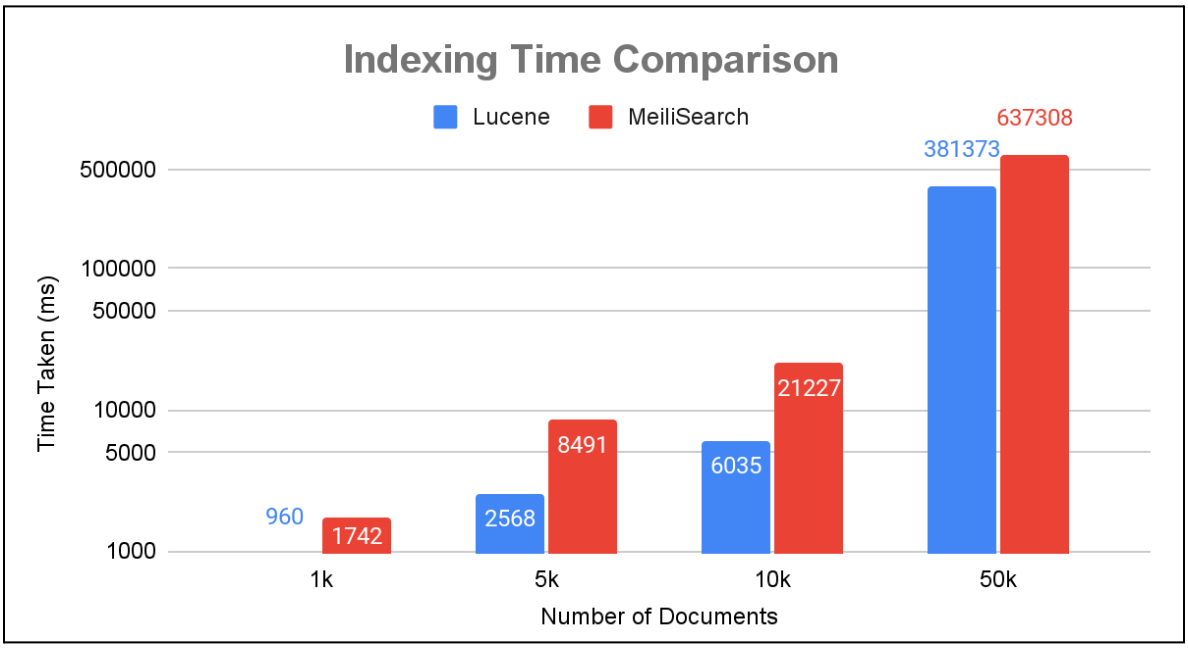

Figure 5.1 Average indexing time comparison between Lucene and MeiliSearch

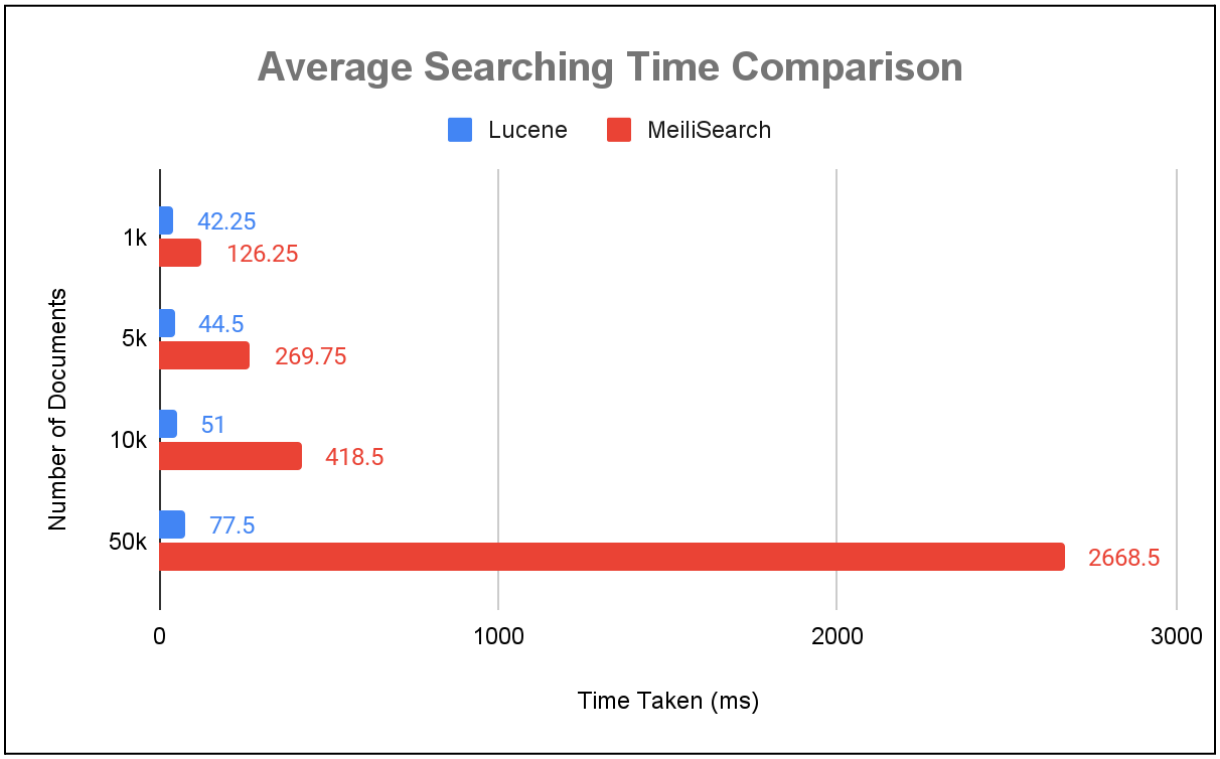

Figure 5.2 Average searching time comparison between Lucene and MeiliSearch

6. Result Discussion

My hypothesis of MeiliSearch taking a longer time to index files than Lucene has proven to be true. However, my other hypothesis of both Lucene and MeiliSearch taking a similar time when a keyword is searched has been proven wrong.

As expected, MeiliSearch took a longer time to index than Lucene and we can see that the time for indexing was not linear. It took more than 30 times to index 50,000 files compared to 10,000 files, while it should have taken 5 times to make it linear. For that reason, the bar chart is logarithm. In the case of Lucene, as the number of data-set increases, indexing time is almost linear.

The indexing time in MeiliSearch is nearly double that of Lucene and it gradually increases as the data set increases. MeiliSearch uses a key-value-based database for indexing and Lucene uses a file system for indexing. Every time a new document is added to the index, updating the database which contains a lot of data consumes more time. Hence, MeiliSearch takes a longer time to index.

I expected that the average search time for both Lucene and MeiliSearch would be similar but this has been proven wrong. When searching for particular keywords, Lucene is way faster with both a small data set and a large data set. From the above diagram, we can see that the search time for Lucene has not changed much even with a large data-set but as the data-set increases, Meili Search takes a longer time to search. The average search time for Lucene hardly changed, whereas the average search time for MeiliSearch increased substantially.

The number of files hit by search in Lucene and Meilisearch differed in some cases. For example, the word “people’s” was not hit by Lucene when searching for the keyword people.

Some words were not hit by Meilisearch because it can index a maximum of 1000 words per attribute (Meili Search, 2021), which in our case is the content of the text files. So if a word occurs after 1000 words MeiliSearch ignores it. To compensate for this limitation we used LimitTokenCountAnalyzer in Lucene and set it to 1000 words. However Lucene also filters out stop-words. So, more than 1000 words may still be included after excluding stop words.

7. Limitations and opportunities for further research

The experiment had a few limitations that could be addressed in further research. Such as the maximum number of documents in our research is limited to about one hundred thousand documents, whereas in the real world the volume can be much larger. Using a larger number of documents in this experiment was not feasible due to the limitation of available computational hardware resources and the time it would consume. This can be extended in further research.

Another area of extending this research is to optimize the code for parallel processing and use upgraded hardware such as multiple processors and memory to take advantage of parallel processing. Indexing and searching time can potentially be reduced with this approach.

We obtained the data files for this experiment only from news articles. We can also do further research on how the technologies behave in data sets from different domains like emails or tweets.

All of these restrictions tackle the question of the generalizability of the research. These limitations were minimized to the best I could. Furthermore, these limitations were present due to the availability of limited resources, time and data available.

8. Conclusion

This experiment aimed to compare Lucene's inverted indexing technology to MeiliSearch for indexing and searching on a large volume of textual data with time complexity in mind. Textual data of different data-set was indexed and searched upon and the results can be found in Figures 5.1 and 5.2.

From these results, I can conclude that as expected MeiliSearch took a longer time to index files than Lucene. However, the indexing time was increasing in a logarithm way for MeiliSearch, while Lucene handled it in an almost linear way. Thus, I can conclude that indexing is better using Lucene for a larger volume of data.

As for the searching time, I expected the search time for Lucene and Meili Search to be similar, but it turned out that Lucene was a lot faster compared to MeiliSearch. Not only that, as the data volume increased, search time remained almost constant for Lucene, while the time taken to search kept increasing while using MeiliSearch.

To answer the research question of this essay, my answer would be that Lucene's inverted index technology is better at both indexing and searching with both a small volume of data-set and a large volume of data-set. As the data volume increased, the effectiveness of Lucene was more evident for both indexing and searching.

Bibliography

Bestari, F. (2019). cnn-news. Retrieved June 8, 2021, from https://www.kaggle.com/fadhrigabestari/cnnnews

Develop Paper. (2020). MO_ On SQL optimization. Retrieved July 9, 2021, from https://developpaper.com/mo_-on-sql-optimization/

elastic. (2021). Stopwords: Performance Versus Precision. Retrieved July 9, 2021, from https://www.elastic.co/guide/en/elasticsearch/guide/current/stopwords.html#stopwords

Ho, R. (2013, February 23). Pragmatic Programming Techniques. Retrieved July 9, 2021, from https://horicky.blogspot.com/2013/02/text-processing-part-2-inverted-index.html

Jain, S. (2018, June 4). *Geeks for geeks*. Difference between Inverted Index and Forward Index. Retrieved July 7, 2021, from https://www.geeksforgeeks.org/difference-inverted-index-forward-index/

Meili Search. (2021, February 11). *MeiliSearch Documentation v0.24*. Tokenization. Retrieved July 9, 2021, from https://docs.meilisearch.com/reference/under_the_hood/tokenization.html#deep-dive-the-meilisearch-tokenizer

Meili Search. (2021, February 11). *MeiliSearch Documentation v0.24*. Known limitations. Retrieved September 18, 2021, from https://docs.meilisearch.com/reference/features/known_limitations.html#design-limitations

Meili Search. (2021, July 20). *MeiliSearch Documentation v0.24*. Concatenate and split queries. Retrieved September 9, 2021, from

https://docs.meilisearch.com/reference/under_the_hood/concat.html#concatenated-queries

Merriam-Webster. (n.d.). typo. Retrieved September 18, 2021, from https://www.merriam-webster.com/dictionary/typo

Stolfi, J. (2009, April 9). A small phone book as a hash table. Retrieved July 9, 2021, from https://en.wikipedia.org/wiki/Hash_table#/media/File:Hash_table_3_1_1_0_1_0_0_SP.svg

Tan, K. (2021). *LuceneTutorial.com*. Retrieved July 9, 2021, from http://www.lucenetutorial.com/basic-concepts.html

Techopedia Inc. (2018, June 13). *Time Complexity*. What Does Time Complexity Mean? Retrieved July 6, 2021, from https://www.techopedia.com/definition/22573/time-complexity

Teddie. (2020, July 2). Exploring Solr Internals : The Lucene Inverted Index. Retrieved July 8, 2021, from https://sease.io/2015/07/exploring-solr-internals-lucene.html

Tutorials Point. (n.d.). Lucene - Indexing Process. Retrieved July 9, 2021, from https://www.tutorialspoint.com/lucene/lucene_indexing_process.htm

Tutorials Point. (n.d.). Lucene - Search Operation. Retrieved July 9, 2021, from https://www.tutorialspoint.com/lucene/lucene_search_operation.htm#:~:text=IndexSearcher%20is%20one%20of%20the,the%20Query%20to%20the%20searcher.

Appendices

Appendix A: Program to index and search with lucene

A.1 IndexBuilder.java

```
/*
 * To change this license header, choose License Headers in Project
Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package com.rijul.searchenginecomparer.lucenestats;

import com.rijul.searchenginecomparer.TextFileFilter;
import java.io.IOException;

/**
 *
 * @author rijul
 */
public class IndexBuilder {

   static String dataDir = "C:\\Users\\rijul\\Desktop\\EE\\Stories";
   Indexer indexer;

   public static void main(String[] args) {
```

```
        IndexBuilder tester;
        dataDir = args[0];
        try {
            tester = new IndexBuilder();
            tester.createIndex(dataDir);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void createIndex(String dataDir) throws IOException {
        String indexDir = dataDir + "_index";
        indexer = new Indexer(indexDir);
        int numIndexed;
        long startTime = System.currentTimeMillis();
        numIndexed = indexer.createIndex(dataDir, new TextFileFilter());
        long endTime = System.currentTimeMillis();
        indexer.close();
        System.out.println(numIndexed+" File indexed, time taken: "
            +(endTime-startTime)+" ms");
    }
}
```

A.2 Indexer.java

```
/*
  * To change this license header, choose License Headers in Project
Properties.
```

```
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package com.rijul.searchenginecomparer.lucenestats;

import com.rijul.searchenginecomparer.CommonConstants;
import java.io.File;
import java.io.FileFilter;
import java.io.FileReader;
import java.io.IOException;
import java.nio.file.Path;

import org.apache.lucene.analysis.standard.StandardAnalyzer;
import org.apache.lucene.document.Document;
import org.apache.lucene.document.Field;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.index.IndexWriterConfig;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;
/**
 *
 * @author rijul
 */
public class Indexer {
    private IndexWriter writer;
```

```
public Indexer(String indexDirectoryPath) throws IOException {
   //this directory will contain the indexes
   Directory indexDirectory =
      FSDirectory.open(Path.of(indexDirectoryPath));

   //create the indexer

   writer = new IndexWriter(indexDirectory,
           new IndexWriterConfig(new StandardAnalyzer()));
}

public void close() throws CorruptIndexException, IOException {
    if (writer.isOpen()) {
         writer.close();
    }
}

private Document getDocument(File file) throws IOException {
   Document document = new Document();

   //index file contents
   Field contentField = new Field(CommonConstants.CONTENTS,
      new FileReader(file));

   //index file name
   Field fileNameField = new Field(CommonConstants.FILE_NAME,
      file.getName(),
```

```
        Field.Store.YES,Field.Index.NOT_ANALYZED);

    //index file path
    Field filePathField = new Field(CommonConstants.FILE_PATH,
        file.getCanonicalPath(),
        Field.Store.YES,Field.Index.NOT_ANALYZED);

    document.add(contentField);
    document.add(fileNameField);
    document.add(filePathField);

    return document;
}

private void indexFile(File file) throws IOException {
    System.out.println("Indexing "+file.getCanonicalPath());
    Document document = getDocument(file);
    writer.addDocument(document);
}

public int createIndex(String dataDirPath, FileFilter filter)
    throws IOException {
    //get all files in the data directory
    File[] files = new File(dataDirPath).listFiles();

    for (File file : files) {
        if(!file.isDirectory()
```

```
            && !file.isHidden()
            && file.exists()
            && file.canRead()
            && filter.accept(file)
         ){
            indexFile(file);
         }
      }
      return writer.numDocs();
   }
}
```

A.3 LuceneSearcher.java

```
/*
  * To change this license header, choose License Headers in Project
Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package com.rijul.searchenginecomparer.lucenestats;

import com.rijul.searchenginecomparer.CommonConstants;
import com.rijul.searchenginecomparer.InputOutputManager;
import java.io.IOException;

import org.apache.lucene.document.Document;
import org.apache.lucene.queryparser.classic.ParseException;
```

```
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopDocs;

/**
 *
 * @author rijul
 */
public class LuceneSearcher {

    String indexDir = "C:\\Users\\rijul\\Desktop\\EE\\stories_index";
    Searcher searcher;

    public static void main(String[] args) {
        LuceneSearcher tester;
        try {
            tester = new LuceneSearcher();
            tester.indexDir = args[0];
                tester.search(new InputOutputManager().InputString("Search
Query:"));
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ParseException pe) {
            pe.printStackTrace();
        }
    }
```

```
         private void search(String searchQuery) throws IOException,
ParseException {
        searcher = new Searcher(indexDir);
        long startTime = System.currentTimeMillis();
        TopDocs hits = searcher.search(searchQuery);
        long endTime = System.currentTimeMillis();

        for (ScoreDoc scoreDoc : hits.scoreDocs) {
            Document doc = searcher.getDocument(scoreDoc);
                                        System.out.println("File:    "    +
doc.get(CommonConstants.FILE_NAME));
        }
        System.out.println(hits.totalHits
                  + " documents found. Time :" + (endTime - startTime) + "
ms");
    }
}
```

A.4 Searcher.java

```
/*
  * To change this license header, choose License Headers in Project
Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package com.rijul.searchenginecomparer.lucenestats;
```

```
import com.rijul.searchenginecomparer.CommonConstants;
import java.io.IOException;

import java.nio.file.Paths;
import org.apache.lucene.analysis.standard.StandardAnalyzer;



import org.apache.lucene.document.Document;
import org.apache.lucene.index.CorruptIndexException;
import org.apache.lucene.index.DirectoryReader;
import org.apache.lucene.index.IndexReader;
import org.apache.lucene.queryparser.classic.ParseException;
import org.apache.lucene.queryparser.classic.QueryParser;

import org.apache.lucene.search.IndexSearcher;
import org.apache.lucene.search.Query;
import org.apache.lucene.search.ScoreDoc;
import org.apache.lucene.search.TopDocs;
import org.apache.lucene.store.Directory;
import org.apache.lucene.store.FSDirectory;



/**
 *
 * @author rijul
 */
public class Searcher {
```

```
    IndexSearcher indexSearcher;
    QueryParser queryParser;
    Query query;

    public Searcher(String indexDirectoryPath) throws IOException {

        indexSearcher = createSearcher(indexDirectoryPath);
             queryParser = new QueryParser(CommonConstants.CONTENTS, new
StandardAnalyzer());
    }

     private static IndexSearcher createSearcher(String indexDir) throws
IOException {
        Directory dir = FSDirectory.open(Paths.get(indexDir));
        IndexReader reader = DirectoryReader.open(dir);
        IndexSearcher searcher = new IndexSearcher(reader);
        return searcher;
    }

    public TopDocs search(String searchQuery)
            throws IOException, ParseException {
        query = queryParser.parse(searchQuery);
        return indexSearcher.search(query, CommonConstants.MAX_SEARCH);
    }

    public Document getDocument(ScoreDoc scoreDoc)
```

```
            throws CorruptIndexException, IOException {
        return indexSearcher.doc(scoreDoc.doc);
    }

}
```

Appendix B: Program to search and index using MeiliSearch

B.1 MeiliIndexer.java

```
/*
 * To change this license header, choose License Headers in Project
Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package com.rijul.searchenginecomparer.meilistats;

import com.meilisearch.sdk.Client;
import com.meilisearch.sdk.Config;
import com.meilisearch.sdk.Index;
import com.rijul.searchenginecomparer.CommonConstants;
import com.rijul.searchenginecomparer.TextFileFilter;
import java.io.File;
import java.io.FileFilter;
import java.io.IOException;
import java.nio.file.Files;
```

```
import java.nio.file.Paths;
import org.json.JSONArray;
import org.json.JSONObject;

/**
 *
 * @author rijul
 */
public class MeiliIndexer {

    public static void main(String[] args) throws Exception {
        if (args.length < 1) {
              System.out.println("Please provide folder to index as first
argument");
            System.exit(0);
        }

        String folderToIndex = args[0];
           Client client = new Client(new Config("http://localhost:7700",
"masterKey"));

        // An index is where the documents are stored
        client.createIndex("stories");
        Index index = client.index("stories");

        String[] displayedFields = {"doc_id", "file_name"};
        index.getSettings().setDisplayedAttributes(displayedFields);
```

```
        long startTime = System.nanoTime();
        IndexFiles(folderToIndex, new TextFileFilter(), index);
        long stopTime = System.nanoTime();
        System.out.println(stopTime - startTime);

    }

     private static void IndexFiles(String folderName, FileFilter filter,
Index index) {
        File folder = new File(folderName);
        File[] listOfFiles = folder.listFiles();
        int fileId = 1;

        JSONArray docList = new JSONArray();
        for (File file : listOfFiles) {
            if (!file.isDirectory()
                    && !file.isHidden()
                    && file.exists()
                    && file.canRead()
                    && filter.accept(file)) {
                String content = readAllBytes(file.getPath());
                docList = new JSONArray();
                JSONObject obj = new JSONObject();
                obj.put(CommonConstants.FILE_ID, fileId++);
                obj.put(CommonConstants.FILE_NAME, file.getName());
                obj.put(CommonConstants.CONTENTS, content);
```

```
            System.out.print("\33[2K\r" + fileId);

            if (fileId % 1000 != 0) {
                docList.put(obj);
            } else {
                IndexJson(docList.toString(), index);
            }
        }
    }
    if (docList.length() > 0) {
        IndexJson(docList.toString(), index);
    }
}

private static void IndexJson(String docListJson, Index index) {
    try {
        index.addDocuments(docListJson);
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}

private static String readAllBytes(String filePath) {
    String content = "";
    try {
        content = new String(Files.readAllBytes(Paths.get(filePath)));
    } catch (IOException e) {
```

```
            e.printStackTrace();
        }

        return content;
    }
}
```

B.2 MeiliSearcher.java

```
/*
 * To change this license header, choose License Headers in Project
Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package com.rijul.searchenginecomparer.meilistats;

import com.meilisearch.sdk.Client;
import com.meilisearch.sdk.Config;
import com.meilisearch.sdk.Index;
import com.meilisearch.sdk.model.SearchResult;
import com.rijul.searchenginecomparer.CommonConstants;

import org.json.JSONArray;
import org.json.JSONObject;

/**
 *
```

```
 * @author rijul
 */
public class MeiliSearcher {

    public static void main(String[] args) {
        if (args.length < 1) {
                System.out.println("Please provide search string as first
argument");
            System.exit(0);
        }
        String searchQuery = args[0];
        try {
                                    //Client   client   =   new   Client(new
Config("http://localhost:7700", "masterKey"));
                                     Client   client   =   new   Client(new
Config("http://192.168.31.179:7700", "masterKey"));
            Index index = client.index("stories");
            // MeiliSearch is typo-tolerant:
            SearchResult results = index.search(searchQuery);
            //System.out.println(results);
            JSONObject obj = new JSONObject(results);
            JSONArray  hits = obj.getJSONArray("hits");
            for (int i = 0; i < hits.length(); i++) {

System.out.println(hits.getJSONObject(i).getString(CommonConstants.FILE_NA
ME));
            }
```

```
            System.out.println();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

}
```

Appendix C: Common utility classes used for both Lucene and MeiliSearch

C.1 CommonConstants.java

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package com.rijul.searchenginecomparer;

/**
 *
 * @author rijul
 */
public class CommonConstants {
```

```
    public static final String CONTENTS = "contents";
    public static final String FILE_NAME = "filename";
    public static final String FILE_PATH = "filepath";
    public static final String FILE_ID = "file_id";
    public static final int MAX_SEARCH = 100000;

}
```

C.2 InputOutputManager.java

```
package com.rijul.searchenginecomparer;

/*
 * To change this license header, choose License Headers in Project
Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
import java.util.Scanner;

/**
 *
 * @author rijul
 */
public class InputOutputManager {

    Scanner sc;
```

```
    public InputOutputManager() {
        sc = new Scanner(System.in);
    }

    public String InputString(String prompt){
        System.out.println(prompt);

        String arg = sc.nextLine();

        return arg;
    }
}
```

C.3 TextFilter.java

```
/*
 * To change this license header, choose License Headers in Project
Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package com.rijul.searchenginecomparer;

import java.io.File;
import java.io.FileFilter;

/**
 *
```

```
 * @author rijul
 */
public class TextFileFilter implements FileFilter {

   @Override
   public boolean accept(File pathname) {
      return pathname.getName().toLowerCase().endsWith(".story");
   }
}
```

Appendix D: Raw data of times obtained

D.1 Indexing time

Indexing Time (ms)		
Number of Documents	**Lucene**	**MeiliSearch**
1k	960	1742
5k	2568	8491
10k	6035	21227
50k	381373	637308

D.2 Average Search time

Average Search time (ms)		
Number of Documents	**Lucene**	**MeiliSearch**
1k	42.25	126.25
5k	44.5	269.75
10k	51	418.5
50k	77.5	2668.5

D.3 Search time 1k

Search Time 1K				
	time taken (ms)		**hits**	
Keyword	**Lucene**	**MeiliSearch**	**Lucene**	**MeiliSearch**
people	37	335	576	575
happy people	42	88	48	46
very happy people	45	62	20	16
very happy sad people	45	20	3	3
	42.25	126.25		

D.4 Search time 5k

Search Time 5K				
	time taken (ms)		hits	
Keyword	Lucene	MeiliSearch	Lucene	MeiliSearch
people	40	750	2873	2863
happy people	44	180	198	181
very happy people	48	116	90	78
very happy sad people	46	33	9	8
	44.5	269.75		

D.5 Search time 10k

Search Time 10K				
	time taken (ms)		hits	
Keyword	Lucene	MeiliSearch	Lucene	MeiliSearch
people	47	1153	5664	5661
happy people	52	285	415	378
very happy people	53	183	217	186
very happy sad people	52	53	19	17
	51	418.5		

D.6 Search time 50k

Search Time 50K				
	time taken (ms)		**hits**	
Keyword	**Lucene**	**MeiliSearch**	**Lucene**	**MeiliSearch**
people	94	9224	28124	28034
happy people	70	638	2124	1924
very happy people	74	543	1177	1002
very happy sad people	72	269	101	83
	77.5	2668.5		