Exploring Apache Lucene: A Deep Dive into Full-Text Search

Kangwei Zhu
kangwei2@illinois.edu
University of Illinois Urbana-Champaign, Urbana, Illinois, USA

1 Proposal

I am planning to conduct a “learning and researching” procedure for the commonly used full-text search engine library — Apache Lucene. The tech review can be splitted into three parts. The first part is going to provide an introduction to full-text search and Lucene. I plan to answer questions like ‘What is full text searching?’ and ‘Why Lucene is a better choice for that”’, and provides both theoretical and benchmark data to consolidate my answer. For the second part, I am going to focus on the theory and implementation(API calling) of the indexing of Lucene. For example, explaining the procedure of how Lucene is able to create the index(inverted index) with some Java code(demo). Notice that the minimalism Lucene environment setup that I learned should be also included prior to this part. Also, I am going to introduce a bit about the Lucene index file format. After that, for the third part, I am planning to explain the querying mechanisms of Lucene. There seems to have multiple types of querying, and I will explain some major types of them with code(demo). Also, I will investigate some methods to optimize the query performance and provide some comparison. I plan to use and select data from the same dataset we were provided for the course assignments in this tech review. If allowed, I might also generate some dummy data(JSON, CSV, text document) by calling the chatbot(ChatGPT API). I plan to spend 5 hours on the first part, 7 hours on the second part, and 9 hours on the third part. My current understanding of Lucene comes from technical books and Lucene documentation. I might find something worthwhile to be covered in my tech review during the learning process. But the above mentioned topics and content is guaranteed to be included in my final report.

2 Summary

For the tech review, I conducted a ’learning while researching’ procedure to the commonly used search engine library **Apache Lucene**. The fantastic CS410 course Lecture 9 - 12 already talked a lot about the concepts of inverted index. Also I learned a lot from completing the machine problem 2. Thus the course content helps me a lot on understanding what Lucene is. Through reading the online documentations of Lucene community, reading those existing papers that are performance and evaluation wide of Lucene [4] [2], and setting up serveral demos locally, I learned what the full-text searching is, why Lucene outperform the traditional databaseswhat do those Lucene terminlogies mean. Also, as I delve deeper, I was suprisingly see that besides the inverted index, the Lucene has so many conjunctions towards this course’s NLP contents, such as scoring, BM25, word-tokenization and stemming, and even nearest neighbor and HWSW problems is supported by Lucene. [7] In short, for this tech review, I am going to start with explaining the concepts of full-text search. Then I’m going to compare the traditional Databases indexing techniques with that of Lucene’s. Also, I’m going to listing and demostrate the commonly seen and used Lucene working scenarios, terminologeis and APIs. Furthermore, I am going to presenting a demo that I set up both Lucene and MySQL locally and comparing their querying performance on large scale dataset. And finally, I’m going to discuss the limitation of Lucene I noticed during my experiment and introduce some novel idea those scholars presented related with Lucene.

3 Introduction

3.1 Full-text Search

Full-text search is a retrieval method that matches all the texts in a document against the search terms. It can extract information such as chapters, sections, paragraphs, sentences, and words from the text as needed. A computer program scans every word in the document and builds an index for each word. Usually the frequency and location of that word in the document is accompanied with the index. When a user performs a query, the system searches using this index, similar to how one looks up a character in a dictionary using a radical index. The major application areas of full-text searching includes: search engines, site-specific search, and e-commerce platforms.

3.2 What is Lucene?

Lucene is an open-source full-text search engine toolkit under Apache [1]. It is not a complete full-text search engine but a framework for building one. It provides a complete query engine and indexing engine, and partial text analysis engine. Lucene aims to provide developers with a simple and easy-to-use toolkit to implement full-text search in their target systems, or to build a full search engine based on it. Lucene offers a simple yet powerful API for indexing and searching full-text. One thing to notice: Luence is not a search engine.

3.3 Why Lucene is commonly used?

Lucene has incredibly ability to search and query the existing data efficiently and timely. On the our hand, traditional Databases commonly use the LIKE keyword for searching, which comes with the following drawbacks:

- It does not leverage efficient indexing methods, so query performance becomes very slow when dealing with large volumes of data.
- The search effectiveness is poor—it can only perform fuzzy matching at the beginning or end of a complete keyword entered by the user. If the user mistypes or adds even one extra character, the results returned can significantly deviate from what the user expected.

Apache Lucene is has better performance because

- Instead of using the indexing approaches that the traditional databases do – using the entire rows of data to perform row scanning, or using the B+ tree to perform data field lookups, Lucene built inverted index, which is tailored for directly

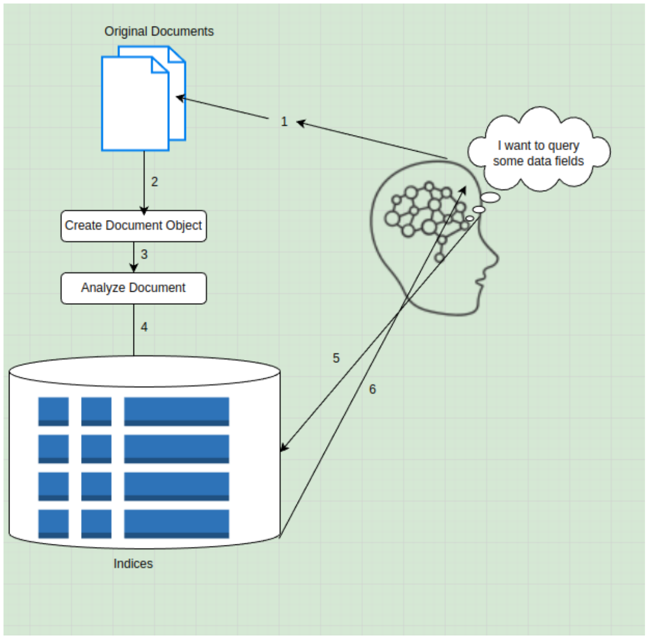


Figure 1: Build Indexes

locating the set of documents containing a specific term. And this feature will significantly imrpove Lucene's query performance, compare with that of traditional DBs'.

- When building the index, Lucene performs tokenization, stop word removal, stemming steps. This will increase the likelihood of matching relevant documents, even if the user inputs incomplete or non-standard keywords.
- Lucene provides fuzzy matching capabilities, such as FuzzyQuery and WildcardQuery, to support typographical errors or minor user input variations, thus enhancing the overall comprehensive searching experience for users. Also, unlike the traditional databases' LIKE queries that merely check for the presence of a term, Lucene scores each result, sorts them by their relevance and only returning the results most likely to match the user's intent.

In the evaluation section later, I will use the NYC Yellow Taxi - Year 2023 [5] dataset, which contains about 38000000 rows of data to comparatively showcase the performance difference between Apache Lucene and the typical relational database: MySQL.

4 Description

4.1 How does the Apache Lucene work?

There is no doubt that one of the most important and commonly used scenarios for Lucene is building an index library.The key question is: how do we create indices? Based on my reading of the documentation and some sample code, I would say there are four main steps involved in creating an index library. As illustrated in the figure above, the process generally follows these steps:

(1) Determine the content to be indexed — First, we need to decide what content we want to make searchable.
(2) Collect documents. We gather the documents that contain the content we wish to index.
(3) Create and analyze document objects — We construct Lucene Document objects and analyze them, which typically involves tokenizing the text into searchable terms.
(4) Store the index — Finally, we save the generated index files to a designated directory.

I think the first two steps are pretty simple. More interesting are steps 3 and 4. As for step 3 of defining and creating a document object, my understanding is the **Lucene Document** is one searchable unit of data. For example, for a search engine, a webpage is one document. Similarly, for an e-commerce site, a product detail page might be one document. A document consists of some **fields**, and each field has data in the form of key-value pairs. For example, a document might have fields such as `"price": "$100"` and `"model": "appliance"`. Notice that the we can name the key as any value we want, as long as we can later on recall it and query it. And In Lucene, every field is configurable with some properties dictating its behavior during indexing and querying time:

- **Stored**: Whether to store and make retrievable the original value of the field. (Using Field.Store.YES/NO)
- **Indexed**: Whether to include the field in the search index.
- **Tokenized and analyzed**: When the field is tokenized and analyzed (e.g., lowercasing, stemming, stopwords removal). After tokenize and analyzed, each token is encapsulated as a `Term` object. A `Term` consists of two parts: the *field name* (e.g., `"content"`) and the *term text* (e.g., `"search"`). It is worth noting that even if the same word appears in multiple fields, each occurrence is treated as a different `Term` due to its field association.

4.2 Common Analyzers and Field Types in Lucene

4.2.1 Analyzers. In Lucene, an `Analyzer` is responsible for processing the text of a field during both indexing and querying. It performs tasks such as tokenization, lowercasing, removing stop words, and applying stemming. Different analyzers are suitable for different languages and use cases. Some commonly used analyzers include:

- **StandardAnalyzer**: Most widely used general-purpose analyzer. Tokenizes, lowercases, and removes a list of English stop words. Ideal for English documents.
- **WhitespaceAnalyzer**: Splits text only by whitespace. It does not lowercase or remove stop words. Useful when token boundaries are strictly space-separated and case-sensitive.
- **SimpleAnalyzer**: Tokenizes text by removing all non-letter characters and lowercases everything. No stop word removal.
- **KeywordAnalyzer**: Treats the entire input as a single token. Useful for fields like IDs, URLs, or exact-match fields.
- **IKAnalyzer** (third-party, Chinese): Chinese language processing is supported. It has fine-grained and intelligent word segmentation, making it the optimal choice for Chinese full-text search.

Besides the above analyzers that contain predefined behavior, Lucene also allows individuals to develop theri **CustomAnalyzer**.

4.2.2 Field Types. A `Field` in Lucene represents a named piece of data in a document. When creating a field, developers must choose an appropriate field type based on how the data should be indexed and stored. Common field types include:

- **TextField**: Used for full-text searchable fields. The content is tokenized and analyzed and is sutiable for fields like `title`, `content`, etc.
- **StringField**: The content for this field is not analyzed which means it's suitable for fields that require exact matching(in contrast of fuzzy), such as `category`, `ID`, ect.
- **StoredField**: It is used to store data that should be retrievable but not public to search, such as the full text of an article for display purposes.
- **IntPoint**, **FloatPoint**, **DoublePoint**, etc.: Special numeric field types used for indexing numeric values. These are not stored by default and are used for range queries or sorting.
- **SortedDocValuesField**: Used for fields that support sorting or faceting, such as `price`, `date`, etc.

Each field must be configured appropriately depending on whether it should be searchable, stored, sortable, or analyzed.

4.3 Choosing the Right Query for the Right Field

In order to make sure that the query engine return the desired result, we have to choose the right query for the right field, which is defined by us during index-building phase. Fail to do so may result the query result to become 0(no hit). For example, use FuzzyQuery to search for a StringField. Since StringField is not analyzed, meaning that no word preprocessing is done, then the Lucene only support **exact match** for that field. Below are some common pairings.

- Use `TermQuery` or `BooleanQuery` with `StringField` for exact matching.
- Use `FuzzyQuery`, `PhraseQuery`, or `WildcardQuery` with `TextField` for natural language search and fuzzy matching.
- Use `RangeQuery` with `IntPoint`, `FloatPoint`, etc., for numeric filtering.

5 Demo: Evaluating MySQL & Lucene's Performance

I've tested all the above mentioned API. In order to verify is Lucene really better than the traditional database in term of handling large scale bussiness logics, I designed a scenario, which is built a small Spring Boot 3 back-end server to simulate the real world taxi company's backend BI scenario that requiring frequent full-text search over the 2023 NYC Yellow-Taxi dataset (37 M trips). I use OpenJDK 17, MySQL 8, Apache Lucene 9.2.2 and running the code on Docker envirnoment, with i9-14900K CPU, 4 CPU 4 Core, 8 core RAM. The README.md and sql/init.sql at the codebase https://github.com/KangweiZhu/lucene-demo provides step by step roadmap that I did to set up the evaluation environment. When the backend server starts, it will Open 4 major endpoints to the localhost, which are **/index/build**, **/benchmark/all**, **/benchmark/all**, **/benchmark/exact**, **/benchmark/fuzzy**. The IndexController is used to build the lucene index. It can be trigger by a post method API calling. We will send the request in the very beginning. And it will take 7.43 minutes to append 37000000 documents to the Lucene's IndexWriter. After indexing is complete, Lucene writes a set of low-level files into the `lucene_index` directory. These files together form an inverted index and are grouped by immutable *segments*. Each file name begins with an underscore and a segment ID (e.g., _a, _b, _d, etc.). A summary of the file types is shown in Table 3.

5.1 Key Steps on Building The Index Index

Listing 1: Index-writer initialisation

```
1  Analyzer analyzer = new KeywordAnalyzer();
2  IndexWriterConfig cfg = new IndexWriterConfig(analyzer);
3  IndexWriter writer = new IndexWriter(
4          MMapDirectory.open(Path.of("lucene_index")),
5          cfg);
```

Explanation: For the Liens 1 to 3, I create a `IndexWriter` backed by a memory-mapped directory and a `KeywordAnalyzer` to ensure the field that requires exact-match will not be tokenized.

Listing 2: Inside the row loop

```
6  Document doc = new Document();
7  doc.add(new StringField("pickup_exact", pickup, Field.Store.
       YES));
8  doc.add(new TextField ("pickup_fuzzy", pickup, Field.Store.
       NO));
9  doc.add(new StoredField("total_amount", rs.getDouble("
       total_amount")));
10 writer.addDocument(doc);
```

Explanation. Lines 7 to 11 creates a Lucene `Document`:

- **Line 8**: `StringField` for exact matching via `TermQuery`.
- **Line 9**: `TextField` for prefix/fuzzy search.
- **Line 10**: `StoredField`—returned to the client, not indexed.

Listing 3: Commit and progress logging

```
11 if (++count % 100_000 == 0) {
12     System.out.printf("written␣%,d%n", count);
13 }
14 writer.commit(); // flush segments and write segments_N
```

Explanation. Lines 13 to 17 print progress every 100k documents and call `writer.commit()` to make the new segments visible to `IndexSearcher`.

The Benchmark API is responsible for handling 3 task:

(1) **Exact Match Testing.** This endpoint accepts one or more pickup zone names and compares the performance of two query methods: a Lucene `TermQuery` on the `pickup_exact` field versus a MySQL query using `WHERE pickup_location = ?`. The number of matching documents and the execution time (in milliseconds) are recorded for both.
(2) **Fuzzy or Prefix Search.** This test evaluates how Lucene's `PrefixQuery` (or optionally, `FuzzyQuery`) performs against

a MySQL `LIKE '%value%'` clause. It measures the total number of partial matches for a given pickup location prefix and reports the runtime of each system.

(3) **Full Count Benchmarking.** This endpoint compares the total document count in the index (using `reader.maxDoc()`) with a MySQL `SELECT COUNT(*)` operation. It highlights Lucene's speed in aggregate operations that do not require full document traversal. The detail statistics could be found at Table 4, below are some key steps in benchmarking, or making the query to both Lucene and MySQL

5.2 Key Benchmark Steps

Listing 4: Exact-match count

```
1 IndexSearcher searcher = new IndexSearcher(
2        DirectoryReader.open(MMapDirectory.open(
             INDEX_PATH)));
3 Query q = new TermQuery(
4        new Term("pickup_exact", pickup.toLowerCase()))
             ;
5 long exactHits = searcher.count(q); // milliseconds
      later
```

Explanation. Lines 1–4 open a read-only `IndexSearcher`, build a case-folded `TermQuery`, and call `count()`. Because the field was indexed with a `KeywordAnalyzer`, Lucene can resolve the hit count in $O(1)$ disk seeks.

Listing 5: Prefix query for fuzzy benchmark

```
6 Query q = new PrefixQuery(
7        new Term("pickup_fuzzy", pickup.toLowerCase()))
             ;
8 long fuzzyHits = searcher.count(q); // 6070 ms for 1.8
       M hits
```

Explanation. Lines 7–8 issue a `PrefixQuery` against the tokenised `pickup_fuzzy` field, measuring how Lucene handles high-recall prefix or fuzzy scenarios.

Listing 6: Full document count

```
9 long totalDocs = reader.maxDoc(); // <1 ms for 37 M
      docs
```

Explanation. Line 11 retrieves the total number of documents directly from the segment metadata; no postings are touched, so the call completes in **sub-millisecond** time even for 37 000 870 records.

5.3 Benchmark Results Summary

Table 1: Full Count Performance Comparison

System	Hits	Time (ms)
Lucene	37,000,870	< 1
MySQL	37,000,870	5,202

Table 2: Prefix/Fuzzy Match for `pickup = "zone-4"`

System	Hits	Time (ms)
Lucene	1,810,893	65
MySQL	1,810,893	12,427

Extension	Purpose
`.si`	Segment info (metadata, doc count, codec, etc.)
`.cfs`	Compound file storage (bundles segment data)
`.cfe`	Compound file entries (file manifest for .cfs)
`.fdt`	Stored field values (payloads for stored fields)
`.fdx`	Offsets into `.fdt` for random access
`.tim`	Terms dictionary (term → postings mapping)
`.tip`	Terms dictionary index (faster lookup into `.tim`)
`.doc`	Postings lists (doc IDs + term frequency)
`.pos`	Positions of each term within each doc
`.nvd`	Norms data (used in scoring, e.g., length norm)
`.nvm`	Norms metadata (field-level descriptions)
`.tmd`	Segment metadata (checksums, skip-lists, etc.)
`segments_N`	Global commit file, listing all active segments

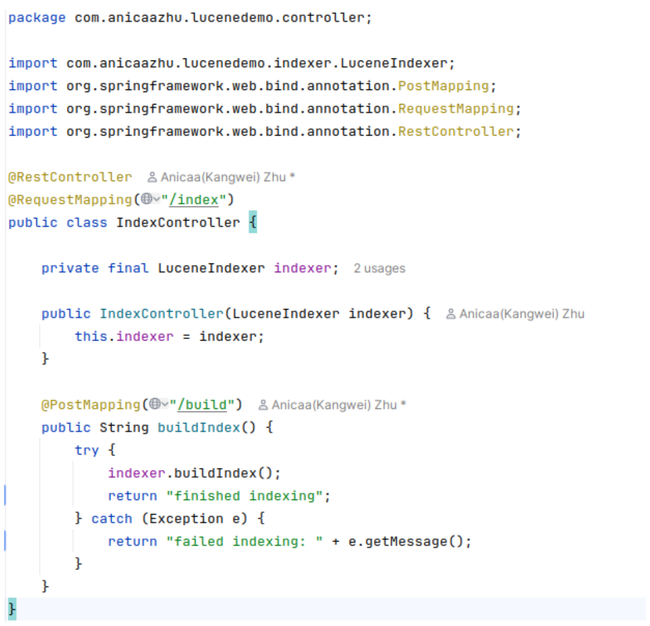
Table 3: Common Lucene index files and their purposes.

Table 4: Exact Match Performance

Pickup Zone	Lucene Hits	Lucene Time (ms)	MySQL Hits	MySQL Time (ms)
zone-45	48,980	1	48,980	10,574
zone-12	16,600	1	16,600	11,163
zone-99	14	<1	14	10,434

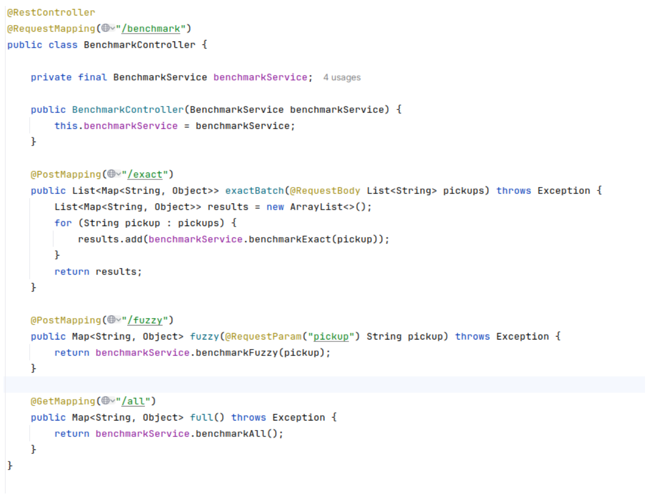
6 Discussion

While Apache Lucene excels at query performance, there is an apparent trade-off: the time it takes to create the index in the first place is extremely slow, especially when ingesting lots of documents. The main reasons for slowness are aggressive tokenization, excessive disk flushing, and asynchronous segment merges. While this is acceptable for static or offline environments, such a cost may be prohibitive for real-time indexing pipelines. Also, doing quality assurance on the ingested dataset should be something worthwhile to note. Espeically on the large scale dataset. Just say the NYC Yellow Taxi Dataset, it contains lots of erronous date format(US date) and null value, which made the index builder just corrupt on the half way and makes the fixing the re-running procedure become super annoying. There have been some work done by scholars, e.g. Su (2020) [6] explored accelerating Lucene's index construction by incorporating a fast suffix sorting algorithm and considering suffix array structures to improve text structuring and reduce preprocessing overhead during indexing. More interestingly, when exploring the approaches of tunning Lucene's query performance, I found IBM researchers proposed a tf-calcaulation method and a better length normalization function in order to improve Lucene's retrieval quality [3] and was approved by Lucene



```java
package com.anicaazhu.lucenedemo.controller;

import com.anicaazhu.lucenedemo.indexer.LuceneIndexer;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController   Anicaa(Kangwei) Zhu *
@RequestMapping("/index")
public class IndexController {

    private final LuceneIndexer indexer;  2 usages

    public IndexController(LuceneIndexer indexer) {  Anicaa(Kangwei) Zhu
        this.indexer = indexer;
    }

    @PostMapping("/build")  Anicaa(Kangwei) Zhu *
    public String buildIndex() {
        try {
            indexer.buildIndex();
            return "finished indexing";
        } catch (Exception e) {
            return "failed indexing: " + e.getMessage();
        }
    }
}
```

Figure 2: IndexController



```java
@RestController
@RequestMapping("/benchmark")
public class BenchmarkController {

    private final BenchmarkService benchmarkService;  4 usages

    public BenchmarkController(BenchmarkService benchmarkService) {
        this.benchmarkService = benchmarkService;
    }

    @PostMapping("/exact")
    public List<Map<String, Object>> exactBatch(@RequestBody List<String> pickups) throws Exception {
        List<Map<String, Object>> results = new ArrayList<>();
        for (String pickup : pickups) {
            results.add(benchmarkService.benchmarkExact(pickup));
        }
        return results;
    }

    @PostMapping("/fuzzy")
    public Map<String, Object> fuzzy(@RequestParam("pickup") String pickup) throws Exception {
        return benchmarkService.benchmarkFuzzy(pickup);
    }

    @GetMapping("/all")
    public Map<String, Object> full() throws Exception {
        return benchmarkService.benchmarkAll();
    }
}
```

Figure 3: BenchmarkController

community https://github.com/apache/lucene/issues/2983 early as 2007.

References

[1] Apache Software Foundation. 2025. Apache Lucene - A High-Performance, Full-Featured Text Search Engine Library. https://lucene.apache.org/. Accessed: 2025-05-07.

[2] Aravind Ayyagiri, Om Goel, and Shalu Jain. 2024. Innovative Approaches to Full-Text Search with Solr and Lucene. *Innovative Research Thoughts* 10 (08 2024), 144–159. https://doi.org/10.36676/irt.v10.i3.1473

[3] Doron Cohen, Einat Amitay, and David Carmel. 2007. Lucene and Juru at TREC 2007: 1-Million Queries Track.

[4] Kerry Koitzsch. 2017. *Advanced Search Techniques with Hadoop, Lucene, and Solr*. 91–136. https://doi.org/10.1007/978-1-4842-1910-2_6

[5] New York City Taxi and Limousine Commission. 2023. 2023 Yellow Taxi Trip Data. https://data.cityofnewyork.us/Transportation/2023-Yellow-Taxi-Trip-Data/4b4i-vvec. Accessed: 2025-05-07.

[6] Ya Su. 2020. *Optimization and Improvement of Lucene Index Algorithm*. 901–907. https://doi.org/10.1007/978-3-030-15235-2_120

[7] Tommaso Teofili and Jimmy Lin. 2019. Lucene for Approximate Nearest-Neighbors Search on Arbitrary Dense Vectors. arXiv:arXiv:1910.10208