Assignment 2: PyLucene

Stig Hellemans, Can Çetinsoy

Background on Lucene

Introduction

Apache Lucene is a high-performance, open-source search library written in Java that provides powerful indexing and searching capabilities. It's widely used to add search functionality to applications, enabling efficient text retrieval, document ranking, and full-text search features. Lucene's core features include the ability to index documents, search and retrieve relevant documents based on queries, and rank results based on different scoring models. It serves as the backbone for many search engines and applications, offering a rich set of tools for both simple and complex search needs. However, Lucene itself is not a stand-alone application. It needs to be embedded within a larger system to function effectively. [1]

Main Functionalities of Lucene

Designed for optimal speed and efficiency, Lucene can process data at over 800GB per hour on modern hardware while keeping memory demands low, requiring only 1MB of heap space. It provides both incremental and batch indexing with equivalent speed, enabling seamless updates without sacrificing performance. Despite its powerful capabilities, the index size is remarkably compact—about 20-30% of the original text size [1]. More about the index size later in 'Analysis'.

Each document is composed of several fields, each containing a certain kind of information, such as title, author, data and content. Lucene employs, besides boolean search, also ranked searching on one of those fields, ensuring that the best matches appear first. A range of advanced query types is available, including phrase, wildcard, proximity, and range queries, offering robust search functionality across fields. High-dimensional vector searches for nearest-neighbor querying are also possible, with flexibility for sorting by any specified field and searching across multiple indexes with combined results. Lucene also offers a variety of enhancements, such as flexible faceting, customizable highlighting, and efficient joins and result grouping. It supports real-time updates and querying, so users can search and index simultaneously. Typo-tolerant, memory-efficient suggesters improve search convenience. [1]

The core of Lucene can be broken down into 3 categories of features: analysis of incoming content and queries, indexing and storage, and searching. More these separate parts below.

Text analysis

Text analysis processes raw text data, transforming it into tokens for indexing. The text analysis chain converts raw text from document fields into a stream of tokens, each represented as a collection of attributes. The primary attribute is the term value, or token itself, but additional attributes can include position, offsets, token type, payload data (stored at a specific position in the index), and custom attributes like part-of-speech tags. The analysis chain starts with character filters, which can remove diacritics or other unwanted characters. Then, tokenizers split text into tokens based on predefined rules, patterns, or language-specific methods (such as whitespace or regex tokenization for languages like Japanese or Chinese). Token filters can then modify tokens for specific use cases, such as stemming, stop word removal, or n-gram generation. This processing chain is adaptable, enabling the creation of complex text analysis pipelines using great collection of tokenizers and token filters available in Lucene. [1, 2]

Indexing and storage in Lucene

Lucene primarily stores two types of indices: inverted indices and forward indices. They each have different use cases.

Index types

1. Inverted Index: Lucene's core index structure is an inverted index, which maps terms to the documents they appear in. This allows quick lookup of documents containing a particular term. For example, if we search for "dog," Lucene can quickly find all documents containing this term. One could incorporate these terms in the index via *Field names*. [1]

2. Forward Index: In Lucene, a forward index stores data by mapping documents directly to their terms. Although not typically used for search, this setup is valuable for enabling features like highlighting and faceting. For example, when searching for "laptop," you might also want to filter or sort results based on document characteristics like price or screen resolution. Moreover, highlighting terms within the original document text (like Google) would be inefficient if it required reconstructing the text by searching all terms associated with a document ID. Instead, linking the original text to the document ID allows for quick access. For row-wise retrieval—accessing all information about a document at once—Stored Field values are used. Alternatively, for efficient retrieval of specific types of document information, column-wise retrievals via Doc values are preferred. [1, 3]

Index construction and storage

Lucene constructs and stores its index through a series of steps, each component designed to support efficient document retrieval. Documents in Lucene are represented as ordered lists of fields. As explained before, fields fall into two main categories: indexed fields, which are searchable, and stored fields, which are retrievable without being searchable. [1]

The IndexWriter class is central to document indexing, managing additions, updates, and deletions. It accomplishes indexing by creating segments, which are independent units within the index. As documents are processed, they undergo tokenization and inversion, and they receive internal identifiers for segment storage. In Lucene, segments are immutable, which allows concurrent reads and updates without locking. Document deletions are managed by marking documents in bitsets instead of immediate removal; these bitsets are applied when segments are accessed, ensuring efficient indexing without recalculating statistics. [1]

Lucene stores indexed data as segments, which are periodically flushed to disk and merged over time to reduce storage and enhance retrieval speed. Each segment contains essential components, such as

term dictionaries (for term lookups), posting lists (tracking term occurrences in documents), and stored fields. Segment merging reduces the overall number of index files and recalculates document statistics, removing data associated with deleted documents. Merging is triggered periodically to ensure optimized storage and search efficiency. [1]

Lucene's Codec API presents the inverted index as a four-dimensional structure with fields for document position and term metadata. The four dimensions—field, term, document, and position—allow for precise traversal and flexible query evaluation. For example, starting at a specific field and term, Lucene can advance to a document and access term frequency and positional data within the document. This structure enables efficient search and retrieval, supporting scalable, high-performance document indexing. The default codec employs variable-byte coding for posting lists and a block tree structure for term dictionaries. This setup achieves a balance of storage efficiency and fast data access. Finally, the Directory API abstracts I/O operations in Lucene, providing a filesystem-like interface that can adapt to various storage solutions, from local filesystems to distributed systems like HDFS. This approach to index construction and storage, which utilizes segments, term dictionaries, posting lists, and effective compression techniques, supports scalable and high-performance document retrieval in Lucene. [2]

Searching

Lucene's search functionality is powered by several components working together to efficiently retrieve and rank documents relevant to a query. The IndexReader provides the ability to access segment information, including term dictionaries and posting lists, facilitating document retrieval by mapping terms to their occurrences across the documents. [1, 2]

The IndexSearcher is the core component for executing search operations. It traverses the inverted index to identify documents containing relevant terms and calculates scores for each document based on various scoring algorithms. For ranking, IndexSearcher relies primarily on the BM25 model, which ranks documents by weighing term frequency and document frequency along with document length normalization. BM25 is the default in Lucene due to its effectiveness for general search use cases. Lucene also supports TF-IDF scoring. [1, 2]

The QueryParser converts user input into structured Query objects, allowing flexible and complex query processing. QueryParser interprets syntax and operators like AND, OR, and NOT, and parses field-specific queries. It tokenizes input using specified analyzers, converting human-readable strings into structured formats, such as BooleanQuery (for combining subqueries), TermQuery (for single-term matches), or PhraseQuery (for exact sequences of terms). This structured format allows IndexSearcher to effectively evaluate queries over indexed documents.

Lucene supports a variety of query types to handle different search requirements [1]:

1. Term Query: Matches documents with a specific term in a given field. To find documents where the field "title" contains the term "Lucene" ⇒ title:Lucene

2. Boolean Query: Combines multiple subqueries using Boolean logic for complex query structures. To find documents where the "title" contains "Lucene" and the "content" includes "search" ⇒ title:Lucene AND content:search

3. Phrase Query: Searches for a sequence of terms in the specified order and proximity. To find documents that contain the exact phrase "machine learning" in the "content" field ⇒ content:"machine learning"

4. Fuzzy Query: Allows for term variations to accommodate misspellings or similar terms. To find documents where the "title" includes terms similar to "Lucen" (e.g., Lucene) ⇒ title:Lucen~1

The ‘1’ stands for fuzziness level. The number of single character edits (insertions, deletions, or substitutions) that are allowed to match the term.

5. Wildcard Query: Uses ***** and **?** as wildcards to match multiple variations of a term, with the asterisk ***** matching zero or more characters and the question mark **?** matching exactly one character. T*o match any word in the "title" that starts with "dev" (such as "developer" or "development") ⇒ title:dev**

6. Range Query: Finds documents within a specific range, such as dates or numbers. To retrieve documents created between 2023 and 2024 ⇒ date:[2023 TO 2024]

7. Prefix Query: Matches terms starting with a given prefix. To find documents where any term in the "title" starts with "auto" ⇒ title:auto*

8. Span Queries: Provides control over term positions, supporting complex phrase and proximity searches. To find documents where "deep" appears within 5 words of "learning" ⇒ spanNear([deep, learning], 5, true)

Lucene also supports **learning-to-rank (LTR)** models, where machine learning algorithms optimize document ranking based on labeled data. This flexibility allows for customized ranking, accommodating a range of scenarios beyond traditional search. [1]

Implementation Details

Constructing the Index

The indexing process in PyLucene is initiated by the *make_database* function, which creates a Lucene index from a given set of documents. Each document is represented by a unique *doc_id* and file path. The process includes the following steps:

1. **Analyzer Selection**: By default, the StandardAnalyzer is used for tokenization and text normalization, though custom analyzers can be provided for more tailored text processing. Several custom analyzers were constructed and tested.
2. **Similarity Configuration**: The similarity function, which determines document scoring, can be set to either BM25Similarity (default) or ClassicSimilarity (TF-IDF).
3. **Directory Management**: The specified index directory is checked for any existing indexes, which are deleted if present, to avoid reusing outdated data. A fresh index directory is then created.
4. **Document Processing**: The *index_txt_file* function is called iteratively for each document, reading the document’s text and adding it to a Lucene Document object via an *IndexWriter*. The *TextField* field is used for the main content, while the *StoredField* field stores the *doc_id* for easy retrieval.
5. **Batch Committing**: To optimize performance, documents are added in batches, committing to disk periodically. This minimizes the frequency of I/O operations, thus accelerating the indexing process.

Searching

Searching in the index is performed by the *query_database* function. This function retrieves the relevant documents for a given query, using an efficient scoring mechanism based on the index configuration. Key steps in this process include:

1. **Analyzer and Similarity Configuration**: The search phase uses the same *Analyzer* and similarity settings (e.g., BM25 or TF-IDF) as the indexing phase to ensure consistency in document scoring.
2. **Query Parsing**: The *QueryParser* processes the raw query string, parsing it into a structured format that Lucene can understand. Any special symbols in the query are removed for compatibility with Lucene's syntax requirements. One example would be the asterisk or question mark used in boolean search.
3. **Executing the Search**: The *IndexSearcher* is used to search the index for matching documents. Using the parsed query, it retrieves the relevant documents, with each document scored based on relevance. The document ID and relevance score for each result are extracted and returned, allowing for further analysis or display.

The code and *results.csv* can be found at this GitHub.

Analysis

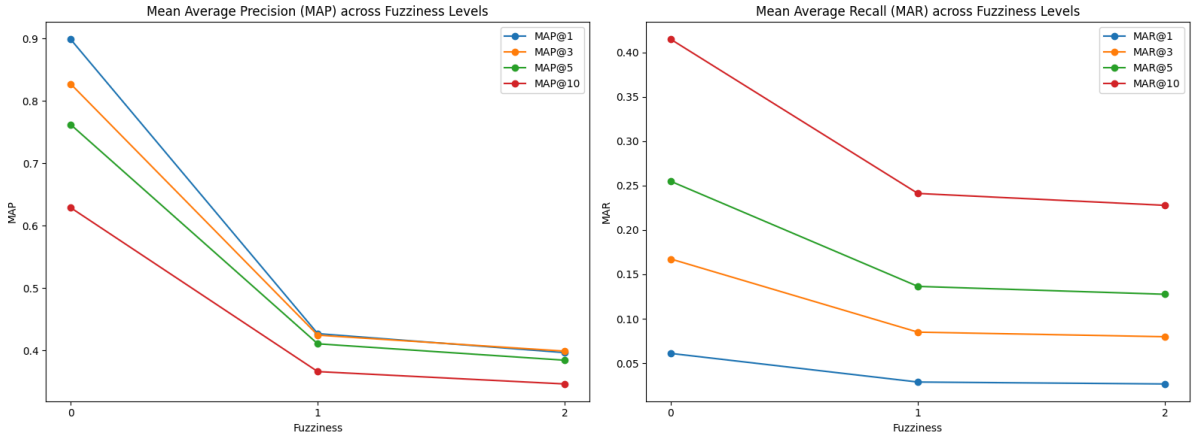
The following table summarizes the experiments worth discussing, with some omitted for practical reasons. For example, an attempt to implement a word-level N-gram analyzer for partial phrase queries, similar to our approach in Assignment 1, resulted in an impractically large vocabulary size. This table presents key evaluation metrics used to compare various implementations. The main performance indicators are MAP@K and MAR@K, while additional metrics include index size, index construction time, vocabulary size, and query times. For Lucene implementations, we distinguish between "original" versions, where the text itself was stored as an additional field, and "no original" versions, which included only the inverted index. It's important to note that only the index construction times for the "original" versions are presented. The "no original" versions completed index construction a few minutes faster.

	Self-implemented		Lucene		
	VSM	VSM + phrase query boosting	Default	TF-IDF	Stemming + Only letters
MAP@1			90.0%	44.4%	85.6%
MAR@1			6.1%	2.9%	5.8%
MAP@3	25.6%	42.1%	82.8%	38.5%	77.6%
MAR@3	5.1%	8.4%	16.8%	7.6%	15.7%
MAP@5			76.3%	35.1%	71.7%
MAR@5			25.5%	11.5%	23.9%
MAP@10	21.3%	32.4%	63.0%	29.4%	59.7%

MAR@10	14.0%	21.3%	41.6%	19.2%	39.3%
Inverted index size (original dataset: 3.51 GB)	1.15 GB	2.32 GB	original: 3.4 GB no original: 1.3 GB	original: 3.4 GB no original: 1.3 GB	original: 2.9 GB no original: 0.83 GB
Index construction time	3.5 h	3.5 h	8 min 37 sec	8 min 20 sec	8 min 15 sec
Query time (ms/query)	44	549	9.3	13	6.6
Vocab size (# terms)	7 180 128	7 180 128	21 411 542	21 423 685	14 809 778

Lucene significantly outperforms the self-implemented version across all aspects. Both index construction and query processing times are an order of magnitude faster, with noticeably better performance overall. It's important to mention that the self-implemented version also incorporated stemming, single-letter word filtering, and stopword exclusion, making the index sizes directly comparable to the Lucene implementation. Notably, the vocabulary size in Lucene is 2–3 times larger. However, this doesn't result in a proportionally larger index size.

Between the Lucene configurations, the default Okapi BM25 model outperforms the older TF-IDF. Interestingly, excluding numbers, special symbols, and applying stemming negatively impacted performance, likely due to some documents containing non-English characters. Finally, we implemented a fuzzy query feature, allowing each token to have a specified fuzziness level to account for spelling variations. However, this reduced performance, as seen in the figure below, and was only feasible for queries up to 50 characters in length.



References

[1] Apache Software Foundation. (n.d.). *Apache Lucene*. Retrieved November 1, 2024, from https://lucene.apache.org/

[2] Bialecki, A., Muir, R., & Ingersoll, G. (2012). Apache Lucene 4. *In Proceedings of the SIGIR Workshop on Open Source Information Retrieval* (OSIR@SIGIR).

[3] Blaszyk, J. (n.d.). Exploring Apache *Lucene Index*. Retrieved November 1, 2024, from https://j.blaszyk.me/tech-blog/exploring-apache-lucene-index/