Inverted Indexes in Lucene

A Technology review Submission

Durga Krovi (dkrovi2@illinois.edu)
Prof: Dr. ChengXiang Zhai
November 7, 2021

Table of Contents

Introduction

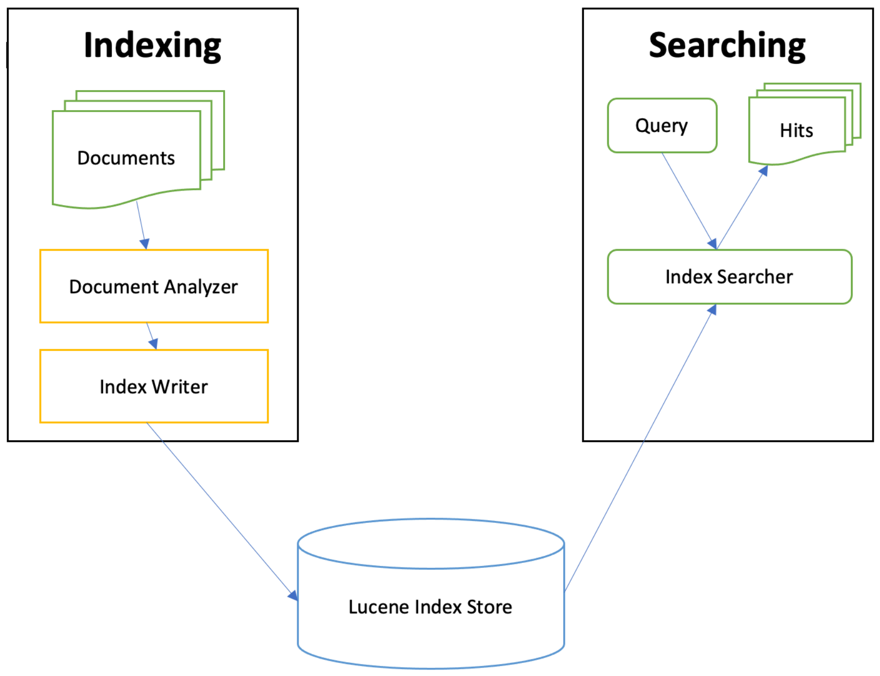
This document aims to dive deep into the inverted index format in Lucene 8.10.1, the latest release (at the time of writing this report) of one of the most popular text indexing and searching library written in Java.

The document is organized as follows:

1. Lucene's High-level Architecture
2. Inverted Index File Format
3. Creation, Maintenance and Search over Inverted Indexes

Lucene Architecture

Apache Lucene is a high-performance, full-featured text search engine library written in Java. The high level architecture of lucene can be described as in the diagram below:



The process of indexing documents includes the following steps:

1. The text present in various types of documents is extracted by the application using the Lucene library.
2. Then, the text is run through an Analyzer that contains various Tokenizers and TokenFilters and is converted into a TokenStream.
3. Then, the token stream is passed to the IndexWriter that updates the underlying index store.

Similarly, the process of searching for documents has the following steps:

1. The terms to search for, are provided as a query.
2. A query builder interprets the query and converts it into a format that Lucene's IndexSearcher can understand.
3. The IndexSearcher searches in the lunch index store and returns the matching documents as Hits.

Inverted Index File Format

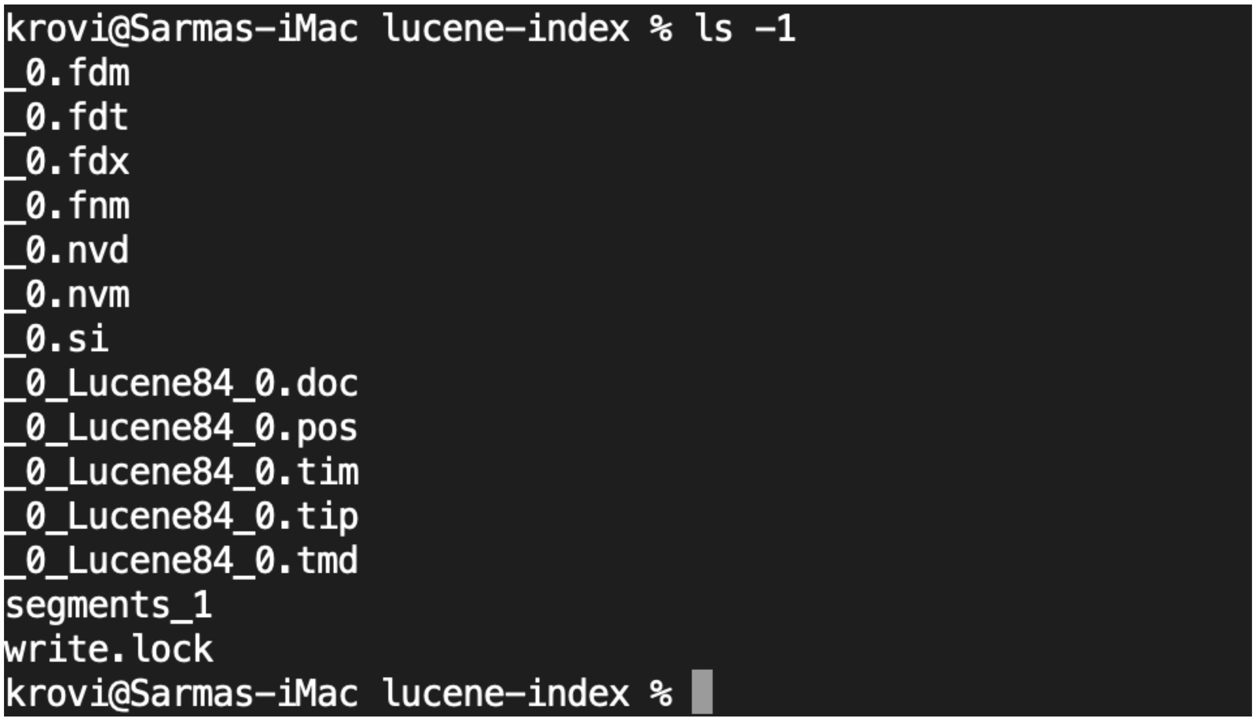
Lucene stores the statistics about the terms in an **Inverted Index** format. The following glossary introduces the key terminology to understand the implementation of inverted index in Lucene.

Glossary

Term	**Description**
Document	A Lucene Document is a sequence of Fields
Field	A Lucene Field is a sequence of Terms
Term	A Lucene Term is a sequence of bytes, fundamental building block of an Index
Segment	A segment is a fully independent index, that can be searched separately
Document Numbers	Zero-based index of numbers used to refer to documents in the index.

At a high-level, an Inverted Index is a collection of sorted integer lists, called posting lists, one for each term that appears in the document corpus. The posting list consists of document identifiers of all the documents containing that term. Each element in the posting list is defined as *Posting*. A Posting can be used to contain other information such as term frequency, positional information and document length. Since the size of an inverted index could be large depending on the document corpus, various compression techniques are used to minimize the storage overhead.

Lucene stores the segment index across various files on the index in uncompressed format. It supports compressed format, by default though. A typical lucene index contains the following files:



```
krovi@Sarmas-iMac lucene-index % ls -1
_0.fdm
_0.fdt
_0.fdx
_0.fnm
_0.nvd
_0.nvm
_0.si
_0_Lucene84_0.doc
_0_Lucene84_0.pos
_0_Lucene84_0.tim
_0_Lucene84_0.tip
_0_Lucene84_0.tmd
segments_1
write.lock
krovi@Sarmas-iMac lucene-index %
```

- **Terms**: Terms are represented in the following files:
  - The ***.tim*** file stores the list of terms in each field along with per-term statistics like document frequency. *Lucene supports pluggable implementations to handle term dictionary through various interfaces, using Service Provider Interface***.**

  - The ***.tip*** file contains an index into the term dictionary for random access.
  - The ***.doc*** file contains the list of documents that contain each term, along with the frequency of the term in that document.
  - The ***.pos*** file contains the list of positions that each term occurs at within documents.
- **Fields**: Stored fields are represented in three files:
  - The ***.fdt*** file, field data file, stores compact representation of documents compressed using LZ4 compression algorithm
  - The ***.fdx*** file, field index file, stores two monotonic arrays, one for the document IDs of each block of compressed documents and another for corresponding offsets on the disk. This organization improves the speed of searching for a document and retrieving it from the disk.
  - The ***.fdm*** file, field meta file, stores metadata about the monotonic arrays stored in the index file.
- **Segments**: The ***.si*** file stores metadata about each segment. This includes the details of the Lucene version that contributed to the segment, number of documents in the index, list of files referred to by this segment etc.,

Lucene also uses other files to store

- per-document normalization values (.nvm, .nvd) like boost factors for documents, fields
- additional scoring factors (.dvd, .dvm) *relevant during searching and ranking*.

Index Creation & Maintenance

As mentioned earlier, a Lucene index contains one or more segment indexes. The IndexWriter component of Lucene creates new segments and flushes them to the directory storage, as new documents are sent in for indexing. Segments are immutable, i.e. updates or deletes create new segments, but do not modify existing segments. The IndexWriter merges groups of smaller segments into single larger ones, to improve the efficiency of search and also to reduce the overall space usage.

One key internal detail of interest is how the document IDs, or docids in short, are handled in a segment and how they are affected during merges. Docids are per-segment sequential values starting from 0. When documents are added to a segment, the docid is incremented and assigned to the new incoming document. When multiple segments are merged, the documents from each segment are assigned new sequential docids.

Index Search

While Lucene offers a wide variety of queries to be run on the index, the TermQuery is of utmost relevance to this document.

TermQuery matches all the documents that contain a specified term, a word, in a certain Field. The TermQuery retrieves all the documents that have the Field specified with the specified term. These matching documents are referred to as Hits in Lucene. These Hits/Documents are scored as described below:

Lucene combines Boolean Model (BM) with Vector Space Model (VSM). The documents approved by BM are scored by VSM. VSM score of document d for query q is the cosine similarity of the weighted query vectors *V(q)* and *V(d)*:

$$\text{cosine-similarity}(q,d) = \frac{V(q) \cdot V(d)}{|V(q)|\ |V(d)|}$$

Then, Lucene refines the VSM score for both search quality and usability:

- Lucene uses a different document length normalization factor to normalize *V(d)* to a vector equal to or larger than the unit vector. This is referred to as *doc-len-normal(d)*. This handles cases where documents contain no duplicated paragraphs.
- Lucene allows assigning a document boost to documents that are more important than others. The score of each document is also multiplied by its boost value, *doc-boost(d)*.
- Lucene allows specifying boost to each query, sub-query and each query term. So, the contribution of query term to the score of a document is multiplied by the boost of that term query term, *query-boost(q)*.

With these, Lucene defines the *Conceptual Scoring Formula* as follows:

$$\text{score}(q,d) = \text{query-boost}(q) \cdot \frac{V(q) \cdot V(d)}{|V(q)|} \cdot \text{doc-len-norm}(d) \cdot \text{doc-boost}(d)$$

This formula simplifies the use case that there is a single field in the index. Lucene derives the **Practical Scoring Function** from this as follows:

$$\text{score}(q,d) = \sum \left(\text{tf}(t \text{ in } d) \cdot \text{idf}(t)^2 \cdot \text{t.getBoost}() \cdot \text{norm}(t,d) \right)$$

Where

- *tf(t in d)* is the term's frequency, computed as

$$\texttt{tf(t in d)} = \text{frequency}^{1/2}$$

- *idf(t)* is the inverse document frequency, to give weightage to rarer terms, computed as

$$\texttt{idf(t)} = 1 + \log \left(\frac{\text{docCount+1}}{\text{docFreq+1}} \right)$$

- t.getBoost() is the search boost of each term “t” in the query “q”
- norm(t, d) is an index-time boost factor calculated from *doc-len-norm(d)* and *doc-boost(d)*. This depends only on the number of tokens in this field in the document such that shorter fields contribute more to the score.

Conclusion

Apache Lucene is a popular search engine library used by many applications (like Apache Solr, Elasticsearch) and technology companies. This library served as a great reference implementation of the core concepts of Text Indexing and Retrieval. The inverted index format used during indexing documents helped understand the complexity involved in creation and maintenance of such indexes for efficient retrieval. The library also supports additional features like boosting documents during indexing and boosting terms in the query during retrieval. The improvised scoring function, while supporting these additional features, explained how its derived from the basic scoring function using the Boolean Model and Vector Space Model.

References

1. Apache Lucene 8.10.1 - Lucene87 file formats
2. Lucene Scoring Functions