Louie Hernandez
NetId: lgherna2
CS410: Text Information Systems

Technology Review: Apache Lucene

Lucene is a powerful open source software library originally written in Java that lets you easily add full search and text indexing as well as information retrieval to applications. Apache Lucene is nearly compatible with any application that requires full text search. Companies across the world use this technology for its high performance, relevancy and scalability. In this article, we will have an overview on Apache Lucene, how it works, and its capabilities.

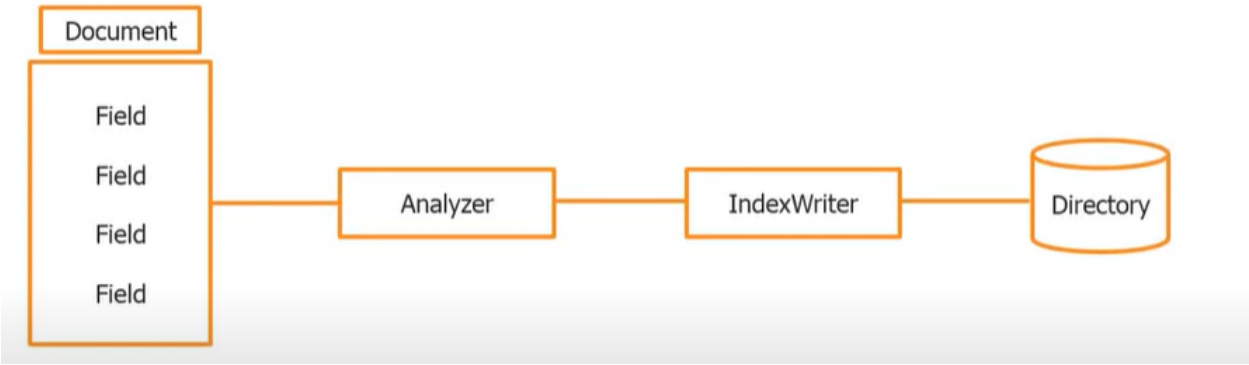
The Lucene search engine indexing collects, parses, and stores data to help facilitate fast and accurate information retrieval. The reason why an index is stored is to help optimize speed and performance in helping find relevant documents in a search query. Without an index, the engine would have to iterate over every document in the collection which would obviously impact performance. It's the difference between creating an index for each document in a collection of 100,000 versus scanning every word in every document of that 100,000 collection.

A document in Lucene terminology is a record, so a record will be initially submitted to the indexing process. When it comes to the workflow for indexing it first needs to be analyzed, the document cannot be submitted as is. Some initial evaluations need to be submitted to the index. Once the analysis is complete the document needs to be converted to tokens or terms. For example, let's say we have a text document that has the phrase "Hello World!", you will first need to break down the entire document into tokens or streams (LinuxHunt). A token is nothing but an independent or atomic unit of text or a stream of text. Once the document is analyzed and converted to tokens or streams, the token or streams are submitted to the indexing process and the document itself is submitted to the inverted index data structure (Medium Kumar). How the index stores those terms is something called a term vector.

Example:

So every term vector will represent a token or a term and when you are searching for any record the lookup happens into this term vector to locate the term or the document where it can be found in the index.

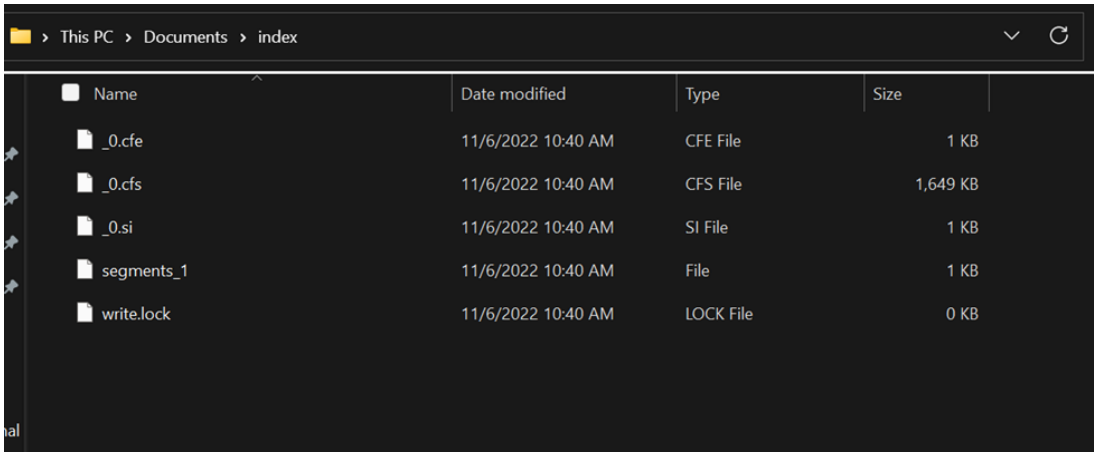
So when we are writing to an index you can have a document or multiple documents submitted to an index, so when we are correlating a document we can use a single record or a set of fields, the document can contain a set of fields. We submit this document to Lucene's analysis process which analyzes the documents and what kind of fields it has, what kind of document it is, and depending on the analyzer you are using there will be some sort of preprocessing. The text goes through various operations of extracting keywords, removing common words and punctuations, and changing words to lowercase (Baeldung). Once the preprocessing is complete, the documents and the fields will be submitted to the IndexWrite, the IndexWriter takes care of writing the index. Once the index is written an index folder will be outputted to the root folder. (Image found in Edureka youtube video)



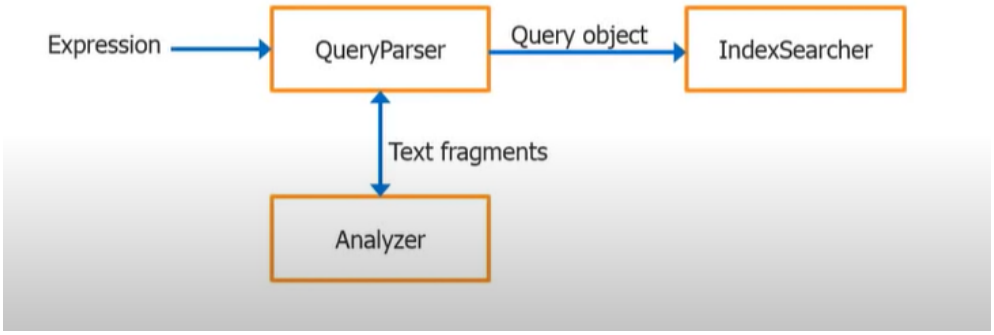
To better understand this, I began to experiment with the source code and some of the demo files from Apache Lucene repository (GitHub URL: https://github.com/apache/lucene.git), and I followed a demo tutorial (https://lucene.apache.org/core/2_9_4/demo.html) to play around with Apache Lucene. For my testing I used the entire collection of Shakespeare's Sonnets as my corpus (https://ocw.mit.edu/ans7870/6/6.006/s08/lecturenotes/files/t8.shakespeare.txt). Once I finished my installation and set up I submitted my document to Lucene's IndexFiles API which calls the IndexWriter and got the following output.

```
louie@BOOK-DESCOR45IL MINGW64 ~/Documents
$ java org.apache.lucene.demo.IndexFiles -docs ./works
Indexing to directory 'index'...
adding .\works\shakespeare_sonnets.txt.txt
Indexed 1 documents in 666 milliseconds
```

As you can see, my document was indexed fairly quickly, and once that was done an index folder was outputted to my root directory.



Once we have indexed the data into an inverted index or the index storage you have multiple ways of retrieving that information back, you can retrieve that information through a Lucene API or a QueryParser.



Query Parser is responsible for translating a textual expression from an end to an arbitrarily complex query for searching. QueryParser works in conjunction with Analyzer to analyze what you are trying to query whether that query fits into that index where we have stored it and the analyzer where we have used it in the indexing process. Once the QueryParser is through scanning the syntax of the expression it submits the query object to the IndexSearcher which retrieves the results of the document.

In the demo project in Lucene there is a SearchFiles API you can use that utilizes both the QueryParser and the IndexSearcher (LuceneApache.Org) . All of Shakespeare's Sonnets can be found in Gutenberg Archives, before all of the works there is a disclaimer at the top, knowing there would be a disclaimer for Gutenberg, I searched for that word and found a match.

```
louie@BOOK-DESCOR45IL MINGW64 ~/Documents
$ java org.apache.lucene.demo.SearchFiles -index ./index -query "Gutenberg"
Searching for: gutenberg
1 total matching documents
1. .\works\shakespeare_sonnets.txt.txt
```

To prove that my implementation is working I searched for a swear word across all of my documents and got zero results.

```
louie@BOOK-DESCOR45IL MINGW64 ~/Documents
$ java org.apache.lucene.demo.SearchFiles -index ./index -query "fuc[redacted]"
Searching for: fucking
0 total matching documents
```

Since the document consists of sonnets I expect certain words to have a higher score than other words found in the document, I expect characters to enter and exit scenes, I also chose a random acronym in the disclaimer to emphasize the difference in scores.
Library of the Future and Shakespeare CDRO

```
louie@BOOK-DESCOR45IL MINGW64 ~/Documents
$ java org.apache.lucene.demo.SearchFiles -index ./index -query "cdroms" -raw
Searching for: cdroms
1 total matching documents
doc=0 score=0.13403125

louie@BOOK-DESCOR45IL MINGW64 ~/Documents
$ java org.apache.lucene.demo.SearchFiles -index ./index -query "scene" -raw
Searching for: scene
1 total matching documents
doc=0 score=0.2872694

louie@BOOK-DESCOR45IL MINGW64 ~/Documents
$ java org.apache.lucene.demo.SearchFiles -index ./index -query "enter" -raw
Searching for: enter
1 total matching documents
doc=0 score=0.2875418
```

Lucene implements a variant of the Tf-Idf scoring model, this model can be explained with the following (Lucene Tutorial):
Tf = term frequency in document (measure of how often a terms appears in the document)
Idf = inverse document frequency (measure of how often appears across the index)
Coord = number of terms in the query that were found in the document
lengthNorm = measure of the importance of a term according to the total number of terms in the field
queryNorm = normalization factor that queries can be compared

If you wanted to, it would be easy to customize the scoring algorithm. Subclass DefaultSimilarity and override the method you want to customize, if you wanted to ignore how common a term appears across the index you could override the `float idf(int i, int i1)` method.

Lucene is a powerful open sourced tool that can easily be used as an indexing and search library but it has some drawbacks. Lucene does not have crawling or HTML parsing. However other projects have come about to extend Apache Lucene's capability. One of those projects is called ElasticSearch, ElasticSearch converts Lucene into a distributed system or search engine

for scaling horizontally. ElasticSearch also provides thread pools, queues, data monitoring API, cluster monitoring API, etc. Elasticsearch hosts data on data nodes. Each data node hosts one or more indices, and each index is divided into shards with each shard holding part of the index's data. Each shard created in Elasticsearch is a separate Lucene instance or process. (Opster). Other interesting Lucene project extensions include Apache Nutch, which provides crawling and HTML parsing; and DocFetcher which is a multiplatform desktop search application.

This article was a quick introduction and demonstration on how to get started with Apache Lucene. I would encourage the reader to look into writing more complex Lucene queries besides a simple term query, such as: PrefixQuery, WildcardQuery, PhraseQuery, FuzzyQuery, BooleanQuery. As well as looking into how to sort search results. If you do this, Apache Lucene will be an even more powerful tool in your arsenal.

References:

https://www.youtube.com/watch?v=vLEvmZ5eEz0

https://www.lucenetutorial.com/advanced-topics/scoring.html

https://www.cnblogs.com/NextNight/p/6928352.html

https://www.baeldung.com/lucene

https://lucene.apache.org/core/9_0_0/demo/

https://linuxhint.com/introduction-to-lucene/

https://medium.com/@karkum/introduction-to-apache-lucene-7d65f67f5231