Technology Review – Apache Lucene

UIUC | CS 410: Text Information Systems | Fall 2022 | Hitesh Yadav

In today's day and world text search is one of the most used features that is used by us and to a great extent has become a part of life. We use text search to gain knowledge about wide variety of topics. We have had a great deal of research and development put into the field and have seen a lot of libraries and projects being built.

This is the technology review is about Apache Lucene full-text search engine API and code library that can easily be used to add search capabilities to applications. Originally written in Java the package is now available in languages like C++, .Net, C, Python, Perl, Ruby to name a few. A detailed Lucene implementation list can be found at this link.

Overview

Apache Lucene packs a lot of features that help build a robust search engine for any application using simple API. Some of the features include:

- Cross-platform (100% pure Java; also available in other languages) [1]
- Scalable, High-Performance Indexing [1]
    - Over 800GB/hour on modern hardware
    - Has less memory footprint with only 1MB Heap
    - Incremental indexing is as fast as batch indexing.
    - Index size is 20-30% the size of text indexed.
- Powerful, Accurate and Efficient Search Algorithms [1]
    - Provides ranked searching that is best result is returned first.
    - The ranking models are pluggable, including Vector Space Model and Okapi BM25
    - Supports many powerful query types like phrase queries, wildcard queries, proximity queries, range queries and more.
    - Supports fielded searching, for example title, author, contents; nearest neighbor search for high-dimensionality vectors; multiple-index searching with merged results; sorting by any field
    - Storage engine can be configured.

Lucene is widely adopted and popular websites. Twitter [2] implemented the earlier version of Apache Lucene to enable the search capabilities. Netflix uses Elasticsearch, which at its core uses Lucene. At its core Lucene API's implement document indexing, specifically creating inverted indexes for the documents using several scoring models which are pluggable. A Lucene document can represent a file on a file system, a row in the database, a new article, a book etc. Each document can have attributes represented a "field" such as, title, author, keyword etc.

Lucene Indexing and Storage

As Lucene relies on indexing the documents to an appropriate internal representation that can be used as needed. An index is composed of sequence of documents, the documents is a sequence of fields, a field is named sequence of terms, a term is sequence of bytes.

Lucene's terms indexes are inverted index, that is, it can, for a given term, list all the documents that contain the term. The index also stores statistics about those terms so that the search can be made efficient. We can also store the text for given fields literally as "field". The text in the field can then be tokenized for indexing or the completed text can be used as index. The indexes can be composed of multiple fully independent index called as sub-indexes or segments. These indexes are evolved by creating a new segment for a newly added document and merging existing segments. Due to this nature of segmented indexes a search may include multiple segments.

The documents in Lucene are referred to using document number, an integer number for the document, with first document numbered 0 and each document added there after gets a number 1 greater than the previous. The document number may also change as the document number within a segment is unique and needs to be updated before using in a larger context such as across segments. [5] The standard technique is to allocate each segment a range of values, based on the range of numbers used in that segment. To convert a document number from a segment to an external value, the segment's *base* document number is added. To convert an external value back to a segment-specific value, the segment is identified by the range that the external value is in, and the segment's base value is subtracted. [5] In other scenario when documents are deleted, gaps are created in the numbering. These are eventually removed as the index evolves through merging. Deleted documents are dropped when segments are merged. A freshly merged segment thus has no gaps in its numbering.

All the files belonging to a segment have the same name with varying extension. When using the Compound File format (default for small segments) these files (except for the Segment info file, the Lock file, and Deleted documents file) are collapsed into a single .cfs file [5]. The different file formats are available here in the official documentation. Lucene also employs a locking mechanism for the files when writing. This is implemented using a write lock file named "write.lock" in the index directory itself ensuring that only one writer is modifying the indexes. If the lock directory is different from the index directory then the write lock will be named "XXXX-write.lock" where XXXX is a unique prefix derived from the full path to the index directory. When this file is present, a writer is currently modifying the index (adding or removing documents).

Lucene search and Scoring [6]

Lucene offers a wide variety of Query implementations. These implementations can be combined in a wide variety of ways to provide complex querying capabilities along with information about where matches took place in the document collection. Once a Query has been created and submitted to the index searcher, the scoring process begins.

Scoring is the heart of Lucene. Lucene indexing is fast, and it hides almost all the complexity from the user. Lucene supports number of pluggable information retrieval models, including:

- Vector Space Models (VSM)
- Probabilistic Models such as Okapi BM25 and DFR
- Language models

These models can be plugged using the Similarity API, and offer extension hooks and parameters for tuning. Scoring is based on how documents are indexed. The Query determines which documents match (a binary decision), while the Similarity determines how to assign scores to the matching documents.

In the typical search application, a Query is passed to the IndexSearcher, beginning the scoring process.
Once inside the IndexSearcher, a Collector is used for the scoring and sorting of the search results. These important objects are involved in a search:

1. The Weight object of the Query. The Weight object is an internal representation of the Query that allows the Query to be reused by the IndexSearcher.
2. The IndexSearcher that initiated the call.
3. A Sort object for specifying how to sort the results if the standard score-based sort method is not desired.

We call one of the search methods of the IndexSearcher, passing in the Weight object created by IndexSearcher and the number of results we want. This method returns a top documents object. The IndexSearcher creates a collector object to collect all the top documents and passes it along with the Weight to another expert search method using a priority queue.

At last, score the documents. The score method takes in the and does scoring. The Scorer that is returned by the Weight object depends on what type of Query was submitted. In most real-world applications with multiple query terms, the Scorer is going to be a Boolean scorer.

Conclusion

Lucene provides variety of methods and pluggable solutioning to create a robust search engine for any application that is platform independent. As in general search is an empirical problem the process will need testing and fitting the parameters in such a way that the process is acceptable by the users. The approach of always be testing is clearly evident in the implementation and needs the code and implementation to evolve continuously for a better outcome.

References

[1] Apache Lucene Features: https://lucene.apache.org/core/features.html
[2] Twitter Engineering Blog: https://blog.twitter.com/engineering/en_us/a/2011/the-engineering-behind-twitter-s-new-search-experience
[3] Netflix Engineering Blog: https://netflixtechblog.com/implementing-the-netflix-media-database-53b5a840b42a
[4] Understanding Lucene: https://dzone.com/refcardz/lucene
[5] Apache Lucene – Index File Formats: https://lucene.apache.org/core/9_4_1/core/org/apache/lucene/codecs/lucene94/package-summary.html#package.description
[6] Apache Lucene – Search and Scoring: https://lucene.apache.org/core/9_4_1/core/org/apache/lucene/search/package-summary.html#package.description
[7] Apache Lucene 4 paper: http://opensearchlab.otago.ac.nz/paper_10.pdf