Abhishek Khare(khare7@illinois.edu)
CS410 Text Information Systems

Text Mining And Analysis Apache Lucene(Searching and Indexing)

Introduction

Apache Lucene is a modern, open source search library designed to provide both relevant results as well as high performance. Lucene is used in a vast range of applications from mobile devices and desktops through Internet scale solutions. Apache Lucene is an open source Java-based search library providing Application Programming Interfaces for performing common search and search related tasks like indexing, querying, highlighting, language analysis and many others. Today, Lucene enjoys widespread adoption, powering search on many of today's most popular websites, applications and devices, such as Twitter, Netflix and Instagram [1, 2, 3] as well as many other search-based applications [4]. Lucene has also spawned several search-based services such as Apache Solr [5] that provide extensions, configuration and infrastructure around Lucene as well as native bindings for programming languages other than Java.

Lucene consists of a number of features that can be broken down into four main categories: analysis of incoming content and queries, indexing and storage, searching, and ancillary modules (everything else). The first three items contribute to what is commonly referred to as the core of Lucene, while the last consists of code libraries that have proven to be useful in solving search-related problems (e.g. result highlighting.)

- **Language Analysis**: The analysis capabilities in Lucene are responsible for taking in content in the form of documents to be indexed or queries to be searched and converting them into an appropriate internal representation that can then be used as needed.
- **Indexing and Storage**:
    - Indexing of user defined documents,
    - Indexing of user defined documents,
    - Lock-free indexing

- **Querying**: Lucene supports a variety of query options, along with the ability to filter, page and sort results as well as perform pseudo relevance feedback.
- **Ancillary Features**: Lucene's ancillary modules contain a variety of capabilities commonly used in building search-based applications. These libraries consist of code that is not seen as critical to the indexing and searching process for all people, but nevertheless useful for many applications. They are packaged separately from the core Lucene library, but are released at the same time as the core and share the core's version number.

Apache Lucene Indexing

Lucene uses the well-known inverted index representation, with additional functionality for keeping adjacent non-inverted data on a per-document basis. Both in-flight and persistent data uses a variety of encoding schemas that affect the size of the index data and the cost of the data compression. Lucene uses pluggable mechanisms for data coding and for the actual storage of index data . Incremental updates are supported and stored in index extents (referred to as "segments") that are periodically merged into larger segments to minimize the total number of index parts [6].

- **Document Model**: Documents are modeled in Lucene as a flat ordered list of fields with content. Fields have name, content data, float weight (used later for scoring), and other attributes, depending on their type, which together determine how the content is processed and represented in the index.
- **Field Types**: There are two broad categories of fields in Lucene documents - those that carry content to be inverted (indexed fields) and those with content to be stored as-is (stored fields). Fields may belong to either or both categories (e.g. with content both to be stored and inverted). Indexed fields can be provided in plain text, in which case it will be first passed through text analysis pipeline, or in its final form of a sequence of tokens with attributes (so called "token stream"). Token streams are then inverted and added to in-memory segments, which are periodically flushed and merged. Depending on the field options, various token attributes (such as positions, starting / ending offsets and per-position payloads) are also stored with the inverted data. A variant of an indexed field is a field where the creation and storage of term frequency vectors was requested. In this case the token stream is used also for building a small inverted index consisting of data from the current field only, and this inverted data is then stored on a per-document and per-field basis.
- **Indexing Chain**: The resulting token stream is finally processed by the indexing chain and the supported attributes (term value, position, offsets and payload data) are added

to the respective posting lists for each term. Also at this stage documents are assigned their internal document Indexing Process identifiers, which are small sequential integers (for efficient delta compression). These identifiers are ephemeral - they are used for identifying document data within a particular segment, so they naturally change after two or more segments are merged (during index compaction).

- **The IndexWriter Class**: The IndexWriter is a high-level class responsible for processing index updates (additions and deletions), recording them in new segments and creating new commit points, and occasionally triggering the index compaction (segment merging). As new documents are being added and in-memory segments are being flushed to storage, periodically an index compaction (merging) is executed in the background that reduces the total number of segments that comprise the whole index. Document deletions are expressed as queries that select (using boolean match) the documents to be deleted. Deletions are also accumulated, applied to the in-memory segments before flushing (while they are still mutable) and also recorded in a commit so that they can be resolved when reading the already flushed immutable segments.
- **The IndexReader Class**: The IndexReader provides high-level methods to retrieve stored fields, term vectors and to traverse the inverted index data. The IndexReader represents the view of an index at a specific point in time. Typically a user obtains an IndexReader from either a commit point (where all data has been written to disk), or directly from IndexWriter (a "near-real time" snapshot that includes both the flushed and the in-memory segments). The IndexReader API follows the composite pattern: an IndexReader representing a specific commit point is actually a list of sub-Readers for each segment. Composed IndexReaders at different points in timeshare underlying subreaders with each other when possible: this allows for efficient representation of multiple point-in-time views. An extreme example of this is the Indexing Process Twitter search engine, where each search operation obtains a new IndexReader.

Apache Lucene Searching

Lucene's primary searching concerns can be broken down into a few key areas: Lucene's query model, query evaluation, scoring and common search extensions.

- **Query Model and Types :** Lucene does not enforce a particular query language: instead it uses Query objects to perform searches. Several Queries are provided as building blocks to express complex queries, and developers can construct their own programmatically or via a Query Parser.Query types provided in Lucene include: term queries that evaluate a single term in a specific field; boolean queries (supporting AND, OR and NOT) where clauses can be any other Query; proximity queries (strict phrase, sloppy phrase that allows for up to N intervening terms) [7,8]; position-based queries (called "spans" in Lucene parlance) that allow to express more complex rules for proximity and relative positions of terms; wildcard, fuzzy and regular expression queries that use automata for evaluating matching terms; disjunction-max query that assigns scores based on the best match for a document across several fields; payload query that processes per-position payload data, etc. Lucene also supports the incorporation of field values into scoring. Named “function queries”, these queries can be used to add useful scoring factors like time and distance into the scoring model. This large collection of predefined queries allows developers to express complex criteria for matching and scoring of documents, in a well-structured tree of query clauses. Typically a search is parsed by a Query Parser into a Query tree, but this is not mandatory: queries can also be generated and combined programmatically. Lucene ships with a number of different query parsers out of the box. Some are based on JavaCC grammars while others are XML based. Details on these query parsers and the framework is beyond the scope of this paper.

- **Query Evaluation:** When a Query is executed, each inverted index segment is processed sequentially for efficiency: it is not necessary to operate on a merged view of the postings lists. For each index segment, the Query generates a Scorer: essentially an enumerator over the matching documents with an additional score() method. Scorers typically score documents with a document-at-a-time (DAAT) strategy, although the commonly used BooleanScorer sometimes uses a TAAT (term-at-a-time)-like strategy when the number of terms is low [9]. Scorers that are “leaf” nodes in the Query tree typically compute the score by passing raw index statistics (such as term frequency) to the Similarity, which is a configurable policy for term ranking. Scorers higher-up in the tree usually operate on sub-scorers, e.g. a Disjunction scorer might compute the sum of its children’s scores. Finally, a Collector is responsible for actually consuming these Scorers and doing something with the results: for example populating a priority queue of the top-N documents [10]. Developers can also implement custom Collectors for advanced use cases such as early termination of queries, faceting, and grouping of similar results.
- **Similarity:** The Similarity class implements a policy for scoring terms and query clauses, taking into account term and global index statistics as well as specifics of a query (e.g. distance between terms of a phrase, number of matching terms in a multi-term query, Levenshtein edit distance of fuzzy terms, etc). Lucene 4 now maintains several per-segment statistics (e.g. total term frequency, unique term count, total document frequency of all terms, etc) to support additional scoring models. As a part of the indexing chain this class is responsible for calculating the field normalization factors (weights) that usually depend on the field length and arbitrary user-specified field boosts. However, the main role of this class is to specify the details of query scoring during query evaluation. As mentioned earlier, Lucene provides several Similarity implementations that offer well-known scoring models: TF/IDF with several different normalizations, BM25, Information-based, Divergence from Randomness, and Language Modeling.
- **Common Search Extensions:** Keyword search is only a part of query execution for many modern search systems. Lucene provides extended query processing capabilities to support easier navigation of search results. The faceting module allows for browsing/drill down capabilities, which is common in many e-commerce applications. Result grouping supports folding related documents (such as those appearing on the same website) into a single combined result. Additional search modules provide support for nested documents, query expansion, and geospatial search.

References

[1] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin, “Earlybird: Real-Time Search at Twitter.”
[2] Interview with Walter Underwood of Netflix. Lucid Imagination. May, 2009. Accessed 6/23/2012.
http://www.lucidimagination.com/devzone/videospodcasts/podcasts/interview-walter-underwood-netflix
[3] Instagram Engineering Blog. Instagram. January 2012. Accessed 6/23/2012.
http://instagramengineering.tumblr.com/post/13649370142/what-powers-instagramhundreds-of-instances-dozens-of
[4] Lucene Powered By Wiki. The Apache Software Foundation. Various. Accessed 6/23/2012.
http://wiki.apache.org/lucenejava/PoweredBy/
[5] Apache Solr. The Apache Software Foundation. Accessed 6/23/2012.
http://lucene.apache.org/solr.
[6] D. Cutting and J. Pedersen. 1989. Optimization for dynamic inverted index maintenance. In Proceedings of the 13th annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR '90), Jean-Luc Vidick (Ed.). ACM, New York, NY, USA, 405-411.
[7] Yves Rasolofo and Jacques Savoy. Term proximity scoring for keyword-based retrieval systems. In Proceedings of the 25th European conference on IR research (ECIR'03), Fabrizio Sebastiani (Ed.). Springer-Verlag, Berlin, Heidelberg, 207-218, 2003.
[8] S. Büttcher, C. Clarke, B. Lushman, B. Term proximity scoring for ad-hoc retrieval on very large text collections. Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval, 621–622, 2006.
[9] Douglass R. Cutting and Jan O. Pedersen, Space Optimizations for Total Ranking, Proceedings of RAIO'97, Computer-Assisted Information Searching on Internet, Quebec, Canada, June 1997, pp. 401-412
[10] A. Moffat, J. Zobel. Fast Ranking in Limited Space. In Proceedings of the Tenth International Conference on Data Engineering. IEEE Computer Society, Washington, DC, USA, 428-437, 1994.