Using Apache Lucene to search text

Easily build search and index capabilities into your applications

In this article, learn about Apache Lucene, the high-performance, full-featured text search-engine library. Explore the Lucene architecture and its core APIs. Learn to use Lucene for cross-platform full-text searching, indexing, displaying results, and extending a search.

Amol Sonawane, a senior software engineer, got his post-graduate diploma in information technology from the International Institute of Information Technology, Bangalore. He has designed and developed software for diverse domains, such as supply-chain management, enterprise application integration, and business intelligence. He has experience developing applications using J2EE technologies and frameworks (Struts and Spring). He resides in Pune, India, with his wife, Anuja. Apart from coding, he enjoys photography, playing chess, and solving puzzles.

18 August 2009
Also available in Japanese Portuguese

Introduction

Lucene is an open source, highly scalable text search-engine library available from the Apache Software Foundation. You can use Lucene in commercial and open source applications. Lucene's powerful APIs focus mainly on text indexing and searching. It can be used to build search capabilities for applications such as e-mail clients, mailing lists, Web searches, database search, etc. Web sites like Wikipedia, TheServerSide, jGuru, and LinkedIn have been powered by Lucene.

Lucene also provides search capabilities for the Eclipse IDE, Nutch (the famous open source Web search engine), and companies such as IBM®, AOL, and Hewlett-Packard. Lucene has been ported to many other programming languages, including Perl, Python, C++, and .NET. As of 30 Jul 2009, the latest version of Lucene in the Java™ programming language is V2.4.1.

Lucene has many features. It:

- Has powerful, accurate, and efficient search algorithms.
- Calculates a score for each document that matches a given query and returns the most relevant documents ranked by the scores.
- Supports many powerful query types, such as PhraseQuery, WildcardQuery, RangeQuery, FuzzyQuery, BooleanQuery, and more.
- Supports parsing of human-entered rich query expressions.
- Allows users to extend the searching behavior using custom sorting, filtering, and query expression

parsing.

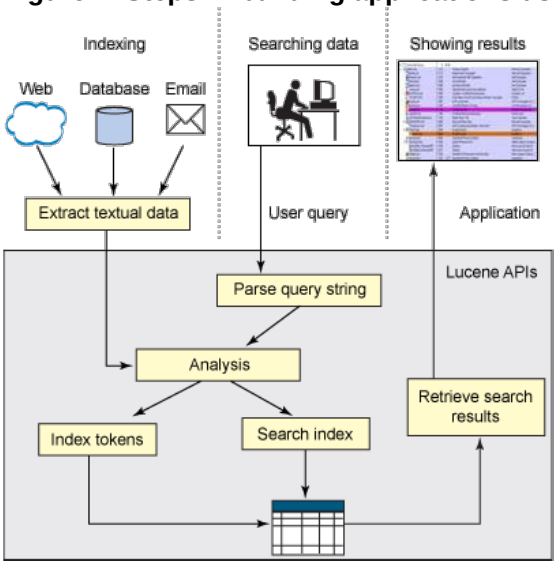
Uses a file-based locking mechanism to prevent concurrent index modifications.

Allows searching and indexing simultaneously.

Building applications using Lucene

As shown in Figure 1, building a full-featured search application using Lucene primarily involves indexing data, searching data, and displaying search results.

Figure 1. Steps in building applications using Lucene



This article uses code snippets from a sample application developed in Java technology using Lucene V2.4.1. The example application indexes a set of e-mail documents stored in properties files and shows how to use Lucene's query APIs to search an index. The example will also familiarize you with basic index operations.

Indexing data

Lucene lets you index any data available in textual format. Lucene can be used with almost any data source as long as textual information can be extracted from it. You can use Lucene to index and search data stored in HTML documents, Microsoft® Word documents, PDF files, and more. The first step in indexing data is to make it available in simple text format. You can do this using custom parsers and data converters.

The indexing process

Indexing is a process of converting text data into a format that facilitates rapid searching. A simple analogy is an index you would find at the end of a book: That index points you to the location of topics that appear in the book.

Lucene stores the input data in a data structure called an *inverted* index, which is stored on the file system or memory as a set of index files. Most Web search engines use an inverted index. It lets users perform fast keyword look-ups and finds the documents that match a given query. Before the text data is

added to the index, it is processed by an analyzer (using an analysis process).

Analysis

Analysis is converting the text data into a fundamental unit of searching, which is called as *term*. During analysis, the text data goes through multiple operations: extracting the words, removing common words, ignoring punctuation, reducing words to root form, changing words to lowercase, etc. Analysis happens just before indexing and query parsing. Analysis converts text data into tokens, and these tokens are added as terms in the Lucene index.

Lucene comes with various built-in analyzers, such as SimpleAnalyzer, StandardAnalyzer, StopAnalyzer, SnowballAnalyzer, and more. These differ in the way they tokenize the text and apply filters. As analysis removes words before indexing, it decreases index size, but it can have a negative effect on precision query processing. You can have more control over the analysis process by creating custom analyzers using basic building blocks provided by Lucene. Table 1 shows some of the built-in analyzers and the way they process data.

Table 1. Lucene's built-in analyzers

Analyzer	Operations done on the text data
WhitespaceAnalyzer	Splits tokens at whitespace
SimpleAnalyzer	Divides text at non-letter characters and puts text in lowercase
StopAnalyzer	Removes stop words (not useful for searching) and puts text in lowercase
StandardAnalyzer	Tokenizes text based on a sophisticated grammar that recognizes: e-mail addresses; acronyms; Chinese, Japanese, and Korean characters; alphanumerics; and more Puts text in lowercase Removes stop words

Core indexing classes

`Directory`

An abstract class that represents the location where index files are stored. There are primarily two subclasses commonly used:

`FSDirectory`— An implementation of `Directory` that stores indexes in the actual file system. This is useful for large indices.

`RAMDirectory`— An implementation that stores all the indices in the memory. This is suitable for smaller indices that can be fully loaded in memory and destroyed when the application terminates. As the index is held in memory, it is comparatively faster.

`Analyzer`

As discussed, the analyzers are responsible for preprocessing the text data and converting it into tokens stored in the index. `IndexWriter` accepts an analyzer used to tokenize data before it is indexed. To index text properly, you should use an analyzer that's appropriate for the language of the text that needs to be indexed.

Default analyzers work well for the English language. There are several other analyzers in the Lucene sandbox, including those for Chinese, Japanese, and Korean.

`IndexDeletionPolicy`

An interface used to implement a policy to customize deletion of stale commits from the index directory. The default deletion policy is `KeepOnlyLastCommitDeletionPolicy`, which keeps only the most recent commits and immediately removes all prior commits after a new commit is done.

`IndexWriter`

A class that either creates or maintains an index. Its constructor accepts a Boolean that determines whether a new index is created or whether an existing index is opened. It provides methods to add, delete, or update documents in the index.

The changes made to the index are initially buffered in the memory and periodically flushed to the index directory. `IndexWriter` exposes several fields that control how indices are buffered in the memory and written to disk. Changes made to the index are not visible to `IndexReader` unless the commit or close method of `IndexWriter` are called. `IndexWriter` creates a lock file for the directory to prevent index corruption by simultaneous index updates. `IndexWriter` lets users specify an optional index deletion policy.

Listing 1. Using Lucene `IndexWriter`

```
//Create instance of Directory where index files will be stored

Directory fsDirectory =  FSDirectory.getDirectory(indexDirectory);

/* Create instance of analyzer, which will be used to tokenize

the input data */

Analyzer standardAnalyzer = new StandardAnalyzer();

//Create a new index

boolean create = true;

//Create the instance of deletion policy

IndexDeletionPolicy deletionPolicy = new KeepOnlyLastCommitDeletionPolicy();

indexWriter =new IndexWriter(fsDirectory,standardAnalyzer,create,

        deletionPolicy,IndexWriter.MaxFieldLength.UNLIMITED);
```

Adding data to an index

There are two classes involved in adding text data to the index.

`Field` represents a piece of data queried or retrieved in a search. The `Field` class encapsulates a field name and its value. Lucene provides options to specify if a field needs to be indexed or analyzed and if its value needs to be stored. These options can be passed while creating a field instance. The table below shows the details of `Field` metadata options.

Table 2. Details of `Field` metadata options

Option	Description
Field.Store.Yes	Used to store the value of fields. Suitable for fields displayed with search results — file path and URL, for example.
Field.Store.No	Field value is not stored — e-mail message body, for example.
Field.Index.No	Suitable for the fields that are not searched — often used with stored fields, such as file path.
Field.Index.ANALYZED	Used for fields indexed and analyzed — e-mail message body and subject, for example.
Field.Index.NOT_ANALYZED	Used for fields that are indexed but not analyzed. It preserves a field's original value in its entirety — dates and personal names, for example.

And a `Document` is a collection of fields. Lucene also supports boosting documents and fields, which is a useful feature if you want to give importance to some of the indexed data. Indexing a text file involves wrapping the text data in fields, creating a document, populating it with fields, and adding the document to the index using `IndexWriter`.

Listing 2 shows an example of adding data to an index.

Listing 2. Adding data to index

```
/*Step 1. Prepare the data for indexing. Extract the data. */

String sender = properties.getProperty("sender");
String date = properties.getProperty("date");
String subject = properties.getProperty("subject");
String message = properties.getProperty("message");
String emaildoc = file.getAbsolutePath();

/* Step 2. Wrap the data in the Fields and add them to a Document */

Field senderField =
        new Field("sender",sender,Field.Store.YES,Field.Index.NOT_ANALYZED);
Field emaildatefield =
                new Field("date",date,Field.Store.NO,Field.Index.NOT_ANALYZED);
Field subjectField =
        new Field("subject",subject,Field.Store.YES,Field.Index.ANALYZED);
Field messagefield =
                        new Field("message",message,Field.Store.NO,Field.Index.ANALYZED);
Field emailDocField =
        new Field("emailDoc",emaildoc,Field.Store.YES,
                                                Field.Index.NO);

Document doc = new Document();
// Add these fields to a Lucene Document
doc.add(senderField);
doc.add(emaildatefield);
doc.add(subjectField);
doc.add(messagefield);
doc.add(emailDocField);

//Step 3: Add this document to Lucene Index.
indexWriter.addDocument(doc);
```

Searching indexed data

Searching is the process of looking for words in the index and finding the documents that contain those words. Building search capabilities using Lucene's search API is a straightforward and easy process. This section discusses the primary classes from the Lucene search API.

Searcher

`Searcher` is an abstract base class that has various overloaded search methods. `IndexSearcher` is a commonly used subclass that allows searching indices stored in a given directory. The `Search` method returns an ordered collection of documents ranked by computed scores. Lucene calculates a score for each of the documents that match a given query. `IndexSearcher` is thread-safe; a single instance can be used by multiple threads concurrently.

Term

Term is the most fundamental unit for searching. It's composed of two elements: the text of the word and the name of the field in which the text occurs. Term objects are also involved in indexing, but they are created by Lucene internals.

Query and subclasses

`Query` is an abstract base class for queries. Searching for a specified word or phrase involves wrapping them in a term, adding the terms to a query object, and passing this query object to `IndexSearcher`'s search method.

Lucene comes with various types of concrete query implementations, such as TermQuery, BooleanQuery, PhraseQuery, PrefixQuery, RangeQuery, MultiTermQuery, FilteredQuery, SpanQuery, etc. The section below discusses primary query classes from Lucene's query API.

`TermQuery`

The most basic query type for searching an index. `TermQuery` can be constructed using a single term. The term value should be case-sensitive, but this is not entirely true. It is important to note that the terms passed for searching should be consistent with the terms produced by the analysis of documents,

because analyzers perform many operations on the original text before building an index.

For example, consider the e-mail subject "Job openings for Java Professionals at Bangalore." Assume you indexed this using the `StandardAnalyzer`. Now if we search for "Java" using `TermQuery`, it would not return anything as this text would have been normalized and put in lowercase by the `StandardAnalyzer`. If we search for the lowercase word "java," it would return all the mail that contains this word in the subject field.

Listing 3. Searching using `TermQuery`

```
//Search mails having the word "java" in the subject field

Searcher indexSearcher = new IndexSearcher(indexDirectory);

Term term = new Term("subject","java");

Query termQuery = new TermQuery(term);

TopDocs topDocs = indexSearcher.search(termQuery,10);
```

`RangeQuery`

You can search within a range using `RangeQuery`. All the terms are arranged lexicographically in the index. Lucene's `RangeQuery` lets users search terms within a range. The range can be specified using a starting term and an ending term, which may be either included or excluded.

Listing 4. Searching within a range

```
/* RangeQuery example:Search mails from 01/06/2009 to 6/06/2009

both inclusive */

Term begin = new Term("date","20090601");

Term end = new Term("date","20090606");

Query query = new RangeQuery(begin, end, true);
```

`PrefixQuery`

You can search by a prefixed word with `PrefixQuery`, which is used to construct a query that matches the documents containing terms that start with a specified word prefix.

Listing 5. Searching using `PrefixQuery`

```
//Search mails having sender field prefixed by the word 'job'

PrefixQuery prefixQuery = new PrefixQuery(new Term("sender","job"));

PrefixQuery query = new PrefixQuery(new Term("sender","job"));
```

`BooleanQuery`

You can construct powerful queries by combining any number of query objects using `BooleanQuery`. It uses `query` and a clause associated with a query that indicates if a query should occur, must occur, or must not occur. In a `BooleanQuery`, the maximum number of clauses is restricted to 1,024 by default. You can set the maximum classes by calling the `setMaxClauseCount` method.

Listing 6. Searching using `BooleanQuery`

```
// Search mails have both 'java' and 'bangalore' in the subject field

Query query1 = new TermQuery(new Term("subject","java"));

Query query2 = new TermQuery(new Term("subject","bangalore"));

BooleanQuery query = new BooleanQuery();
```

```
query.add(query1,BooleanClause.Occur.MUST);

query.add(query2,BooleanClause.Occur.MUST);
```

PhraseQuery

You can search by phrase using `PhraseQuery`. A `PhraseQuery` matches documents containing a particular sequence of terms. `PhraseQuery` uses positional information of the term that is stored in an index. The distance between the terms that are considered to be matched is called *slop*. By default the value of slop is zero, and it can be set by calling the `setSlop` method. `PhraseQuery` also supports multiple term phrases.

Listing 7. Searching using PhraseQuery

```
/* PhraseQuery example: Search mails that have phrase 'job opening j2ee'

   in the subject field.*/

PhraseQuery query = new PhraseQuery();

query.setSlop(1);

query.add(new Term("subject","job"));

query.add(new Term("subject","opening"));

query.add(new Term("subject","j2ee"));
```

WildcardQuery

A `WildcardQuery` implements a wild-card search query, which lets you do searches such as arch* (letting you find documents containing architect, architecture, etc.). Two standard wild cards are used:

* for zero or more

? for one or more

There could be a performance drop if you try to search using a pattern in the beginning of a wild-card query, as all the terms in the index will be queried to find matching documents.

Listing 8. Searching using WildcardQuery

```
//Search for 'arch*' to find e-mail messages that have word 'architect' in the subject

field./

Query query = new WildcardQuery(new Term("subject","arch*"));
```

FuzzyQuery

You can search for similar terms with `FuzzyQuery`, which matches words that are similar to your specified word. The similarity measurement is based on the Levenshtein (edit distance) algorithm. In Listing 9, `FuzzyQuery` is used to find a close match of a misspelled word "admnistrtor," though this word is not indexed.

Listing 9. Searching using FuzzyQuery

```
/* Search for emails that have word similar to 'admnistrtor' in the

subject field. Note we have misspelled admnistrtor here.*/

Query query = new FuzzyQuery(new Term("subject", "admnistrtor"));
```

QueryParser

`QueryParser` is useful for parsing human-entered query strings. You can use it to parse user-entered query expressions into a Lucene query object, which can be passed to `IndexSearcher`'s search

method. It can parse rich query expressions. `QueryParser` internally converts a human-entered query string into one of the concrete query subclasses. You need to escape special characters such as `*`, `?` with a backslash (`\`). You can construct Boolean queries textually using the operators `AND`, `OR`, and `NOT`.

Listing 10. Searching for human-entered query expression

```
QueryParser queryParser = new QueryParser("subject",new StandardAnalyzer());

// Search for emails that contain the words 'job openings' and '.net' and 'pune'

Query query = queryParser.parse("job openings AND .net AND pune");
```

Displaying search results

`IndexSearcher` returns an array of references to ranked search results, such as documents that match a given query. You can decide the number of top search results that need to be retrieved by specifying it in the `IndexSearcher`'s search method. Customized paging can be built on top of this. You can add a custom Web application or desktop application to display search results. Primary classes involved in retrieving the search results are `ScoreDoc` and `TopDocs`.

`ScoreDoc`

A simple pointer to a document contained in the search results. This encapsulates the position of a document in the index and the score computed by Lucene.

`TopDocs`

Encapsulates the total number of search results and an array of `ScoreDoc`.

The code snippet below shows how to retrieve documents contained in the search results.

Listing 11. Displaying search results

```
/* First parameter is the query to be executed and
   second parameter indicates the no of search results to fetch */
   TopDocs topDocs = indexSearcher.search(query,20);
   System.out.println("Total hits "+topDocs.totalHits);

   // Get an array of references to matched documents
   ScoreDoc[] scoreDosArray = topDocs.scoreDocs;
   for(ScoreDoc scoredoc: scoreDosArray){
      //Retrieve the matched document and show relevant details
      Document doc = indexSearcher.doc(scoredoc.doc);
      System.out.println("\nSender: "+doc.getField("sender").stringValue());
      System.out.println("Subject: "+doc.getField("subject").stringValue());
      System.out.println("Email file location: "
                                      +doc.getField("emailDoc").stringValue());
   }
```

Basic index operations

Basic index operations include removing and boosting documents.

Removing documents from an index

Applications often need to update the index with the latest data and remove older data. For example, in the case of Web search engines, the index needs to be updated regularly as new Web pages get added and non-existent Web pages need to be removed. Lucene provides the `IndexReader` interface that lets you perform these operations on an index.

`IndexReader` is an abstract class that provides various methods to access the index. Lucene internally refers to documents with document numbers that can change as the documents are added to or deleted from the index. The document number is used to access a document in the index. `IndexReader` cannot be used to update indices in a directory for which `IndexWriter` is already opened. `IndexReader` always searches the snapshot of the index when it is opened. Any changes to the index are not visible until

`IndexReader` is reopened. It is important that applications using Lucene reopen their `IndexReaders` to see the latest index updates.

Listing 12. Deleting documents from index

```
// Delete all the mails from the index received in May 2009.
IndexReader indexReader = IndexReader.open(indexDirectory);
indexReader.deleteDocuments(new Term("month","05"));
//close associate index files and save deletions to disk
indexReader.close();
```

Boosting documents and fields

Sometimes you might want to give more importance to some of the indexed data. You can do so by setting a boost factor for a document or a field. By default, all the documents and fields have the same default boost factor of 1.0.

Listing 13. Boosting fields

```
if(subject.toLowerCase().indexOf("pune") != -1){
// Display search results that contain pune in their subject first by setting boost factor
        subjectField.setBoost(2.2F);
}
//Display search results that contain 'job' in their sender email address
if(sender.toLowerCase().indexOf("job")!=-1){
        luceneDocument.setBoost(2.1F);
}
```

Extending the search

Lucene provides the advanced feature called *sorting*. You can sort search results by fields that indicate the relative position of the documents in the index. The field used for sorting must be indexed but not tokenized. There are four possible kinds of term values that may be put into sorting fields: integers, longs, floats, or strings.

Search results can also be sorted by index order. Lucene sorts the results by decreasing relevance, such as computed score by default. Sorting order can also be changed.

Listing 14. Sorting search results

```
/* Search mails having the word 'job' in subject and return results
   sorted by sender's email in descending order.
 */
SortField sortField = new SortField("sender", true);
Sort sortBySender = new Sort(sortField);
WildcardQuery query = new WildcardQuery(new Term("subject","job*"));
TopFieldDocs topFieldDocs =
                        indexSearcher.search(query,null,20,sortBySender);
//Sorting by index order
topFieldDocs = indexSearcher.search(query,null,20,Sort.INDEXORDER);
```

Filtering is a process that constrains the search space and allows only a subset of documents to be considered for search hits. You can use this feature to implement search-within-search results, or to implement security on top of search results. Lucene comes with various built-in filters such as BooleanFilter, CachingWrapperFilter, ChainedFilter, DuplicateFilter, PrefixFilter, QueryWrapperFilter, RangeFilter, RemoteCachingWrapperFilter, SpanFilter, etc. `Filter` can be passed to `IndexSearcher`'s search method to filter documents that match the filter criteria.

Listing 15. Filtering search results

```
/*Filter the results to show only mails that have sender field
prefixed with 'jobs' */
Term prefix = new Term("sender","jobs");
Filter prefixFilter = new PrefixFilter(prefix);
WildcardQuery query = new WildcardQuery(new Term("subject","job*"));
indexSearcher.search(query,prefixFilter,20);
```

Conclusion

Lucene, a very popular open source search library from Apache, provides powerful indexing and

searching capabilities for applications. It provides a simple and easy-to-use API that requires minimal understanding of the internals of indexing and searching. In this article, you learned about Lucene architecture and its core APIs.

Lucene has powered various search applications being used by many well-known Web sites and organizations. It has been ported to many other programming languages. Lucene has a large and active technical user community. If you're looking for an easy-to-use, scalable, and high performing open-source search library, Apache Lucene is a great choice.

Download

Description	Name	Size
Lucene code sample	os-apache-lucenesearch-SampleApplication.zip	755KB

Resources

Learn

Learn all about Apache Lucene, including the latest news.

Lucene in Action, by Erik Hatcher and Otis Gospodnetic, is the authoritative guide to Lucene. It describes how to index your data, including types you definitely need to know such as MS Word, PDF, HTML, and XML. It introduces you to searching, sorting, filtering, and highlighting search results.

To listen to interesting interviews and discussions for software developers, check out developerWorks podcasts.

Stay current with developerWorks' Technical events and webcasts.

Follow developerWorks on Twitter.

Check out upcoming conferences, trade shows, webcasts, and other Events around the world that are of interest to IBM open source developers.

Visit the developerWorks Open source zone for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.

Watch and learn about IBM and open source technologies and product functions with the no-cost developerWorks On demand demos.

Get products and technologies

Innovate your next open source development project with IBM trial software, available for download or on DVD.

Download IBM product evaluation versions or explore the online trials in the IBM SOA Sandbox and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

Participate in developerWorks blogs and get involved in the developerWorks

Dig deeper into Open source on developerWorks

- Overview
- New to Open source
- Projects
- Technical library (tutorials and more)
- Forums
- Events

community.