Programagic

Programming Ideas & Experiments (/)

Lucene.Net's Core Indexing and Search Classes

Aug 9, 2016

The previous article (../../../../blog/basics-of-information-retrieval-using-lucene) of this series discussed the basics of information retrieval using Lucene. This article will look at a few important Lucene.Net classes (version 3.0.3 at the time of writing) that are at the core of indexing and search followed by a simple search application.

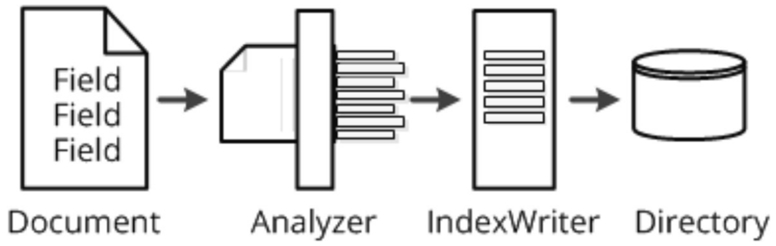
Core Indexing Classes

There are several classes the participate in the indexing process:

1. `Field` : the building block of a `Document` . It has a name, a value and a series of options that control how the value is stored and treated during the indexing process. Note that a document can have multiple fields with the same name.
2. `Document` : a representation of the basic unit of indexing and search, such as an email message or a web page, that needs to be made retrievable for future use. It consists of a collection of `Field` .
3. `Analyzer` : responsible to extract meaningful terms out of the provided text to build the index.
4. `IndexWriter` : the central component that can open an existing index or create a new one and add, update or delete `Document` in it.
5. `Directory` : an abstract class that represents the location where the Lucene index is stored.

The following diagram illustrates the roles of these classes in the indexing process.

Classes involved in the indexing process



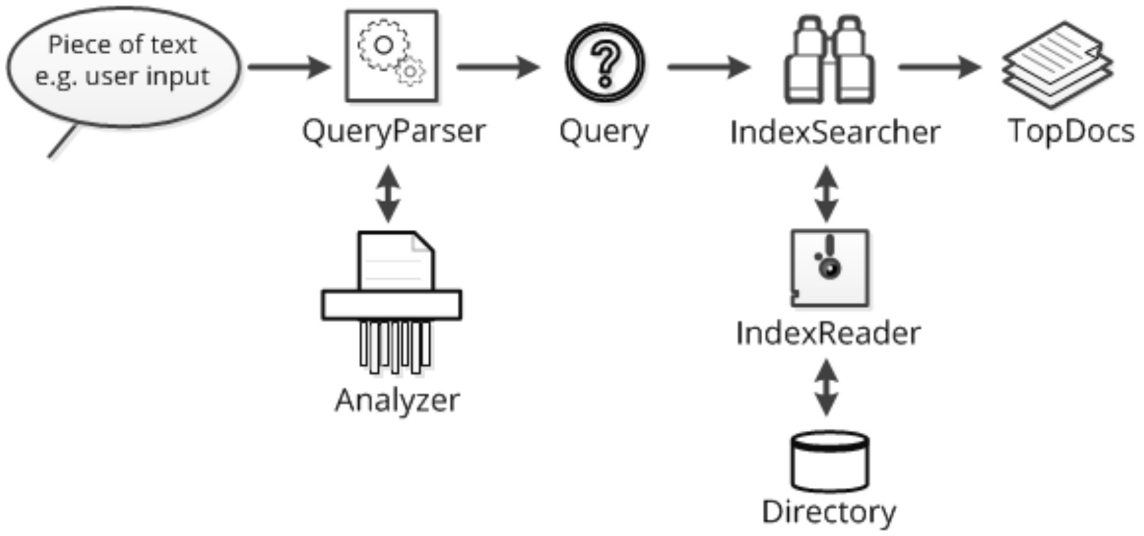
Core Search Classes

The search api to perform a basic search involves a few classes.

1. `Term` : a field-name field-value pair that is the basic unit for searching (similar to `Field`). Note that `Term` objects are also created during indexing but they are usually hidden the Lucene internal mechanics.
2. `Query` : an abstract class that is used to probe the index to find matching documents. Lucene provides a number of concrete implementations that cater for specific use cases.
3. `QueryParser` : generates a query from provided text and using a specified `Analyzer` . While it is possible to create instances of `Query` subclasses, `QueryParser` is often a convenient and handy alternative.
4. `IndexReader` : an abstract class that provides access to an index.
5. `IndexSearcher` : a lightweight wrapper around an `IndexReader` that provides search functionality. Opening an `IndexReader` is a relatively expensive operation, `IndexSearcher` has less overhead and multiple instances can reuse the same underlying `IndexReader` .
6. `TopDocs` : a list of pointers (document id) to matching documents that are part of the search result. The client application will loop over the `TopDocs` to load each `Document` for building the desired output.

The following diagram illustrates the roles of these classes during search.

Classes involved during search



A Simple Indexing and Search Application

BasicLuceneApp is a simple Lucene.Net demo movie search program that contains enough information to demonstrate the structure of a search application. It does the following:

- Creates a Lucene.Net index from a list of movies on startup (location is hard coded to C:\Temp\Index directory in this case)
- Prompts the user to specify a search term in plain text or a query in Lucene syntax (https://lucene.apache.org/core/2_9_4/queryparsersyntax.html)
- Searches the index to find documents against the specified field that match the input criteria (value)
- Lists all the movies in the search results along with their relevancy score

Generating the Movie Index

The movie index is generated by creating a Lucene.Net document for each movie in the list and adding it to the index using IndexWriter's `AddDocument` method. Each document contains a set of fields with specified title, value and options. There are several constructors available for Field class, the one used in this example specifies options for storing and indexing. Storing options (`Store.YES` or `Store.NO`) determine whether the value can be stored for later retrieval during searching.

Indexing options (`Field.Index.*`) control how the field's value is broken down and made searchable. The available indexing options are:

- `Index.NO` : field value is not available for searching
- `Index.NOT_ANALYZED` : field is available for searching, but not broken down, the entire value is stored as a single token and is suitable for items such as website url, postal codes
- `Index.NOT_ANALYZED_NO_NORMS` : same as NOT_ANALYZED, but uses less space and memory as the norms (factors that influence the relevance score) information is not stored
- `Index.ANALYZED` : field's value is broken down into separate searchable tokens - useful for longer pieces of texts such as titles and summary
- `Index.ANALYZED_NO_NORMS` : similar to ANALYZED without norms information to save index and memory

The steps to generate the index are:

1. Create a `Directory`
2. Create an `Analyzer` - choose one that suits the needs of the application
3. Create an `IndexWriter`
4. Create a `Document` for each source object (movie) and add appropriate `Field` information to it
5. Add the `Document` to the index
6. Commit the index

```
private static string indexLocation = "C:\\Temp\\Index";
private static Directory directory = FSDirectory.Open(indexLocation);
private static Analyzer analyzer = new StandardAnalyzer(Lucene.Net.Util.Version.LUCENE_30);

private static void GenerateMovieIndex()
{
    DeleteIndexFiles(indexLocation);
    using (IndexWriter writer = new IndexWriter(directory, analyzer, IndexWriter.MaxFieldLength.UNLIMITED))
    {
        foreach (var movie in MovieDatabase.GetMovies())
            writer.AddDocument(CreateMovieDocument(movie));
        writer.Commit();
    }
}
private static void DeleteIndexFiles(string indexLocation)
{
    foreach (var file in System.IO.Directory.GetFiles(indexLocation))
        System.IO.File.Delete(file);
}

private static Document CreateMovieDocument(Movie movie)
{
    Document document = new Document();
    document.Add(new Field("genre", movie.Genre, Field.Store.YES, Field.Index.NOT_ANALYZED));
    document.Add(new Field("title", movie.Title, Field.Store.YES, Field.Index.ANALYZED));
    document.Add(new Field("summary", movie.Summary, Field.Store.YES, Field.Index.ANALYZED));
    document.Add(new Field("year", movie.Year.ToString(), Field.Store.YES, Field.Index.NOT_ANALYZED));

    foreach (var actor in movie.Actors)
        document.Add(new Field("actor", actor, Field.Store.YES, Field.Index.ANALYZED));

    return document;
}
```

Searching the Index

Searching involves creating a `Query` and executing it with an `IndexSearcher` . There are many types of queries which address specific use cases - for example TermQuery, MultiTermQuery, NumericRangeQuery, WildcardQuery etc. Choosing the right query type is important to get desired results. In this demo, a `QueryParser` is used to generate a query from user's input.

The steps to search the index are:

1. Open the `Directory`
2. Create an `IndexSearcher` using the `Directory (this creates an IndexReader under the hoods)`
3. Create a `Query`
4. Invoke `IndexSearcher.Search` method to get `TopDocs` as search results
5. Load matching documents from `TopDocs.ScoreDocs` and create the desired output

```
private static string indexLocation = "C:\\Temp\\Index";
private static Directory directory = FSDirectory.Open(indexLocation);
private static Analyzer analyzer = new StandardAnalyzer(Lucene.Net.Util.Version.LUCENE_30);

private static void Search()
{
    IndexSearcher searcher = new IndexSearcher(directory);
    QueryParser parser = new QueryParser(Lucene.Net.Util.Version.LUCENE_30, "title", analyzer);

    while (true)
    {
        Console.Write("\nSearch:>");
        FindMatchingDocuments(searcher, parser, Console.ReadLine());
    }
}

private static void FindMatchingDocuments(IndexSearcher searcher, QueryParser parser, string input)
{
    if (string.IsNullOrWhiteSpace(input) || input.Trim().ToLowerInvariant().Equals("exit")) Environment.Exit(0);

    Query query = parser.Parse(input);
    TopDocs docs = searcher.Search(query, 10);
    PrintResults(searcher, docs);
}

private static void PrintResults(IndexSearcher searcher, TopDocs docs)
{
    Console.WriteLine($"{docs.TotalHits} movie(s) found");

    for (int i = 0; i & lt; docs.TotalHits; i++)
    {
        var document = searcher.Doc(docs.ScoreDocs[i].Doc);
        Console.WriteLine($"{(i + 1)}: {document.Get("title")} ({document.Get("year")}) [{docs.ScoreDocs[i].Score}]");
    }
}
```

Executing Queries

Search is conducted against the "title" field by default (as specified in the `QueryParser` constructor). However it is possible to specify a different field at runtime and influence the search behaviour using Lucene query syntax. Running the application can help in understanding certain concepts and how Lucene works. Note the output will be in the format `rank: (year) title [score]` and will be sorted by the highest score.

Term Query

Find the whole term "twins" in the default search field (title)

```
search:>twins
Number of movies found: 1
1: (1988) Twins [2.252763]
```

Find "twin". This does not return any result as the whole term is not matched.

```
search:>twin
Number of movies found: 0
```

Find "twins" or "terminator" in the default field.

```
search:>twins terminator
Number of movies found: 3
1: (1988) Twins [0.8709884]
2: (1984) The Terminator [0.5856731]
3: (2015) Terminator Genisys [0.3660457]
```

Find "twins" or "terminator" in the default field, expressed differently (syntax `field:value`), but same result.

```
search:>title:twins OR title:terminator
Number of movies found: 3
1: (1988) Twins [0.8709884]
2: (1984) The Terminator [0.5856731]
3: (2015) Terminator Genisys [0.3660457]
```

Find all Arnold's movies.

```
search:>actor:arnold
Number of movies found: 3
1: (1984) The Terminator [0.5848559]
2: (1988) Twins [0.5848559]
3: (2015) Terminator Genisys [0.5848559]
```

Find all movies with actor=arnold AND year = 1984 (AND is case sensitive).

```
search:>actor:arnold AND year:1984
Number of movies found: 1
1: (1984) The Terminator [2.185107]
```

Some of the queries above could also be expressed using the TermQuery class.

```
 Term term = new Term(field, value);
 TermQuery query = new TermQuery(term);
```

Wildcard Query

Find movie titles that start with "t". Use the star sign to match any number of characters.

```
search:>title:t*
Number of movies found: 3
1: (1984) The Terminator [1]
2: (1988) Twins [1]
3: (2015) Terminator Genisys [1]
```

A question mark can also be used, but it matches a single character

```
search:>t?ins
Number of movies found: 1
1: (1988) Twins [1]
```

Note that "*" or "?" cannot be used as the first character of the search term.

Wildcard queries can also be expressed using the `WildcardQuery` class.

```
 Term term = new Term(field, value);
 Query query = new WildcardQuery(term);
```

MAY Contain, MUST Contain, MUST NOT Contain

Any query prefixed with the plus sign (+) requires that term to exist in the specified field. Any query prefixed with the minus sign (-) excludes documents that contain the search term in the specified field.

Find all movies, MUST HAVE actor=arnold and MUST HAVE "genisys" in the title.

```
search:>+actor:arnold +genisys
Number of movies found: 1
1: (2015) Terminator Genisys [1.490532]
```

Find all movies: MUST HAVE actor=arnold and MAY HAVE "genisys" in the title. Note the higher score for "terminator genisys".

```
search:>+actor:arnold genisys
Number of movies found: 3
1: (2015) Terminator Genisys [1.490532]
2: (1984) The Terminator [0.1664537]
3: (1988) Twins [0.1664537]
```

Find all movies: MUST HAVE actor=arnold but which MUST NOT HAVE "genisys" in the title.

```
search:>+actor:arnold -title:genisys
Number of movies found: 2
1: (1984) The Terminator [0.5848559]
2: (1988) Twins [0.5848559]
```

Range Query

Find movies between 2000 and 2015

```
search:>year:[2000 TO 2015]
Number of movies found: 3
1: (2000) Gladiator [1]
2: (2013) The Wolf of Wall Street [1]
3: (2015) Terminator Genisys [1]
```

Fuzzy Query

Lucene allows to perform fuzzy searches. This is a very powerful aprroximation technique that finds results which can be relevant to the search term even though they do not exactly correspond to it. For example, goat and coat. One use case is the "did you mean"" feature employed by search engines. For example, if a user incorrectly writes "Torontor", search engines like Google show "Did you mean: Toronto" along with the results.

Find movies where the title somewhat looks like "Walt". Use the tilde sign "~".

```
search:>Walt~
Number of movies found: 1
1: (2013) The Wolf of Wall Street [1.126382]
```

Proximity Query

The tilde sign can also be used for proximity searches. For example, find movie titles with the word wolf and street within 5 words of each other.

```
search:>"wolf street"~5
Number of movies found: 1
1: (2013) The Wolf of Wall Street [1.300633]
```

Find movie titles with the word wolf and street within 1 word of each other.

```
search:>"wolf street"~1
Number of movies found: 0
```

Boosting and the Relative Importance of Terms

If there is more than one clause in a query, it is possible to give more importance to one clause over the other. This is called boosting and affects the relevance.

Find all movie titles starting with "t", but treat those having "genisys" in the title as more important (boosted 5 times in this example using caret (^) symbol). Note the score.

```
search:>t* genisys^5
Number of movies found: 3
1: (2015) Terminator Genisys [1.490893]
2: (1984) The Terminator [0.04421602]
3: (1988) Twins [0.04421602]
```

Consequences of Using an Analyzer

Lucene has many different analyzers, each with a different behaviour. In this demo, the StandardAnalyzer is used and it has the following consequences:

- The search is case insensitive.

  ```
  search:>TERMINATOR
  Number of movies found: 2
  1: (1984) The Terminator [1.847298]
  2: (2015) Terminator Genisys [1.154561]
  ```

- Common English grammar words are treated as noise by StandardAnalyzer and omitted. Hence "the", "of" are not part of the index. These words are contained in a list called stopwords.

```
search:>the
Number of movies found: 0

search:>of
Number of movies found: 0
```

Final Thoughts

There is a lot more to Lucene queries than can be discussed in a single post, but I hope this article provides an idea of the possibilities that Lucene can open up. I would also recommend Luke (https://code.google.com/archive/p/luke/), which is a handy development and diagnostic tool that can read from and write to existing indexes and allows to modify their content.

The source code is available on github (https://github.com/r15h1/lucenedemo/tree/master/src/Lucene01QueryConsole). The next article (../../../../blog/lucene-analysis-process) will take a deeper dive into the analysis process.