Programagic

Programming Ideas & Experiments (/)

Search with Lucene.Net 4.8 (Part 1) - Basic Implementation

Jun 26, 2017

Lucene.Net is getting a significant upgrade and is now available as a series of Nuget (https://www.nuget.org/packages/Lucene.Net/4.8.0-beta00001) packages thanks to the commendable efforts of Synnhersko et al (http://code972.com/blog/2016/12/104-lucene-net-4-8-a-pre-release-introduction-video). The new version (4.8) supports .Net Core which makes it easier to add search functionalities into .Net applications targetted for different platforms. This article discusses a basic implementation of a search application using Lucene.Net 4.8.

For a quick introduction to Lucene, please refer to these:

1. The Basics of Information Retrieval using Lucene (../../../../../blog/basics-of-information-retrieval-using-lucene)
2. Lucene.Net's Core Indexing and Search Classes (../../../../../blog/lucene-core-indexing-and-search-classes)
3. Lucene's Analysis Process (../../../../../blog/lucene-analysis-process)

The source code is available from github (https://github.com/r15h1/lucene.net-experiments/tree/01-basic-implementation).

Data Source

The data used comes from the pagila (https://github.com/devrimgunduz/pagila) postgresql movie database. For convenience, a select part of it was exported as a json file (free download (https://github.com/r15h1/lucene.net-experiments), available under the same bsd (https://opensource.org/licenses/bsd-license.php) license as pagila). The movies are loaded as a list of `Movie` objects

Movie and related classes

```
public class Movie
{
    public int MovieId { get; set; }
    public string Title { get; set; }
    public string Description { get; set; }
    public string Rating { get; set; }
    public List<Actor> Actors { get; set; } = new List<Actor>();
    public List<Category> Categories { get; set; } = new List<Category>();
}

public class Actor
{
    public int ActorId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Category
{
    public int CategoryId { get; set; }
    public string CategoryName { get; set; }
}
```

Generating the Index

The data has to be loaded into Lucene as documents to make it searchable. A Lucene document is a set of fields that has a name and a value (contents). There are different types of fields in Lucene.Net 4.8, the common ones are listed below:

- `StringField` : stored as a single token, not broken down but made searchable, option to store original data
- `TextField` : content is broken down into separate tokens and made searchable, option to store original data
- `StoredField` : content is not searchable, meant only to store data that can be retrieved unaltered

The Lucene index - made up of documents - is stored in a directory. In this demo, the file system directory (`FSDirectory`) will be used to commit the index as a set of files on disk with the help of an `IndexWriter` .

Salient code snippet to generate an index

```
public void BuildIndex(IEnumerable movies)
{
    if (movies == null) throw new ArgumentNullException();

    foreach (var movie in movies)
    {
        Document movieDocument = BuildDocument(movie);
        writer.UpdateDocument(new Term("id", movie.MovieId.ToString()), movieDocument);
    }

    writer.Flush(true, true);
    writer.Commit();
}

private Document BuildDocument(Movie movie)
{
    Document doc = new Document
    {
        new StoredField("movieid", movie.MovieId),
        new TextField("title", movie.Title, Field.Store.YES),
        new TextField("description", movie.Description, Field.Store.NO),
        new StoredField("snippet", MakeSnippet(movie.Description)),
        new StringField("rating", movie.Rating, Field.Store.YES)
    };

    return doc;
}

private string MakeSnippet(string description)
{
    return (string.IsNullOrWhiteSpace(description) || description.Length <= SNIPPET_LENGTH)
        ? description
        : $"{description.Substring(0, SNIPPET_LENGTH)}...";
}
```

`writer.UpdateDocument` identifies a document by its id and adds it to the index. If the document already exists in the index, it is first deleted and

then added.

In this example, an assumption is made that the movie description field is huge. To save space, it will be made searchable but not stored for later retrieval. At search time, the actual value displayed in the search results will show a snippet of up to 100 characters which is stored in the snippet `StoredField`. Note that only title and description fields are searchable.

Lucene.Net 4.8 contains an array of useful analyzers to break down text. A single analyzer can be used to break down data in all indexabe fields in a document or a different analyzer can be specified on a per field basis using the `PerFieldAnalyzerWrapper` class. Analyzers can also be created on the fly anonymously. In this demo, the `StandardAnalyzer`, which is useful in many cases, will be used.

StandardAnalyzer

```
private Analyzer SetupAnalyzer() => new StandardAnalyzer(MATCH_LUCENE_VERSION, StandardAnalyzer.STOP_WORDS_SET);
```

A set of stop words is provided to remove some common English words that are not useful for searching (e.g. the, an, is etc.)

Searching the Index

The search term is parsed using a `MultiFieldQueryParser` (acting on title and description fields) to generate a Lucene query. The search is then carried out with an `IndexSearcher`. Lucene 4.8 provides a `SearcherManager` utility class to safely share `IndexSearcher` instances across multiple threads.

Search related code

```
private QueryParser SetupQueryParser(Analyzer analyzer)
{
    return new MultiFieldQueryParser (
        MATCH_LUCENE_VERSION,
        new[] { "title", "description" },
        analyzer
    );
}

public SearchResults Search(string queryString)
{
    int resultsPerPage = 10;
    Query query = BuildQuery(queryString);
    searchManager.MaybeRefreshBlocking();
    IndexSearcher searcher = searchManager.Acquire();

    try
    {
        TopDocs topdDocs = searcher.Search(query, resultsPerPage);
        return CompileResults(searcher, topdDocs);
    }
    finally
    {
        searchManager.Release(searcher);
        searcher = null;
    }
}

private Query BuildQuery(string queryString) =>  queryParser.Parse(queryString);

private SearchResults CompileResults(IndexSearcher searcher, TopDocs topdDocs)
{
    SearchResults searchResults = new SearchResults() { TotalHits = topdDocs.TotalHits };
    foreach (var result in topdDocs.ScoreDocs)
    {
        Document document = searcher.Doc(result.Doc);
        Hit searchResult = new Hit
        {
```

```
            Rating = document.GetField("rating")?.GetStringValue(),
            MovieId = document.GetField("movieid")?.GetStringValue(),
            Score = result.Score,
            Title = document.GetField("title")?.GetStringValue(),
            Snippet = document.GetField("snippet")?.GetStringValue()
        };

        searchResults.Hits.Add(searchResult);
    }

    return searchResults;
}
```

Displaying Results

The solution consists of a console application (cli) and a search class library (searchlib). When the application is run, the user is prompted to type the search term. When the search is completed, the cli displays the title, the score and a snippet of the description.

The search term can also be expressed in Lucene query syntax (with symbols such as *, +, ?, ~ etc.). Happy Searching!

Search results in console app