Search with Lucene.Net 4.8 (Part 2) - A Few Tricks and Tips

Jul 21, 2017

Recently our search engine was upgraded to Lucene.Net 4.8 (originally 3.3) which presented the right window of opportunity to perform some fine tuning and refactoring. Careful planning is often required to suit the indexing and querying strategies to specific scenarios, assumptions and content (bilingual English and French in our case). Below are a few tricks and tips that worked in our case - an ASP.Net restful search api wrapper around Lucene.Net.

Some of the ideas discussed here were inspired from this amazing post (http://blog.swwomm.com/2013/07/tuning-lucene-to-get-most-relevant.html) and adapted to our reality.

Leverage SearcherManager in Multi-Threaded Scenarios

Initially in our Lucene.Net 3.3 implementation, a single IndexSearcher per index was shared among multiple threads / requests in our api. Lucene.Net 4.8 provides a simpler way to handle multiple threads searching an index through the built-in `SearcherManager` class . Calling its `Acquire` method returns an IndexSearcher ready to be used. Once the search is completed, calling its `Release` method and setting the instance to null avoids it being used again to load stored documents.

SearcherManager class usage

```
private readonly IndexWriter writer;
private readonly SearcherManager searchManager;
private readonly Analyzer analyzer;
private const LuceneVersion MATCH_LUCENE_VERSION= LuceneVersion.LUCENE_48;

void Initialize(string indexPath)
{
    analyzer = new EnhancedEnglishAnalyzer(MATCH_LUCENE_VERSION, EnglishAnalyzer.DefaultStopSet);
    writer = new IndexWriter(FSDirectory.Open(indexPath), new IndexWriterConfig(MATCH_LUCENE_VERSION, analyzer)
            {
                OpenMode = OpenMode.CREATE_OR_APPEND
            });
    searchManager = new SearcherManager(writer, true, null);
}

SearchResults Search(Query query, int resultsPerPage)
{
    IndexSearcher searcher = searchManager.Acquire();
    try
    {
        TopDocs topdDocs = searcher.Search(query, resultsPerPage);
        return CompileResults(searcher, topdDocs);
    }
    finally
    {
        searchManager.Release(searcher);
        searcher = null;
    }
}
```

Customize the Lucene Query

Most end users expect relevant, up-to-date and contextual results and have no clue about Lucene query syntax. We found out that using a `QueryParser` with Lucene's query syntax was more complicated than using `Query` classes to build a custom query from users' input. A custom query was needed to improve the perception of search relevancy in users' eyes.

In our indexing strategy, synonyms are applied to offset the need for special treatment of abbreviations and jargon at query-time. Before

building the query, user's input is first broken up into a list of tokens.

Splitting user's input

```
IList<string> Tokenize(string userInput)
{
    List<string> tokens = new List<string>();
    using(var reader = new StringReader(userInput))
    using(TokenStream stream = analyzer.GetTokenStream("myfield", reader))
    {
        stream.Reset();
        while(stream.IncrementToken())
            tokens.Add(stream.GetAttribute<ICharTermAttribute>().ToString());
    }
    return tokens;
}
```

A `BooleanQuery` is then built with the following:

- If the number of tokens from user input is greater than one:
  - Include a heavily boosted exact `PhraseQuery` with a small slop on the main field
  - Include a nested `BooleanQuery` with a `MinimumNumberShouldMatch` on the main field - only first 5 tokens considered if user input is long
- Always include `TermQuery` instances for each token on every field

A simplified version of custom query creation

```
class FieldDefinition
{
    public string Name { get; set; }
    public bool IsDefault { get; set; } = false;
    //other properties omitted
}

//in a different class
Query BuildQuery(string userInput, IEnumerable<FieldDefinition> fields)
{
    BooleanQuery query = new BooleanQuery();
    IList<string> tokens = Tokenize(userInput);

    //combine tokens present in user input
    if(tokens.Count > 1)
    {
        FieldDefinition defaultField = fields.FirstOrDefault(f => f.IsDefault == true);
        query.Add(BuildExactPhraseQuery(tokens, defaultField), Occur.SHOULD);

        foreach(var q in GetIncrementalMatchQuery(tokens, defaultField))
            query.Add(q, Occur.SHOULD);
    }

    //create a term query per field - non boosted
    foreach(var token in tokens)
        foreach(var field in fields)
            query.Add(new TermQuery(new Term(field.Name, token)), Occur.SHOULD);

    return query;
}

Query BuildExactPhraseQuery(IList<string> tokens, FieldDefinition field)
{
    //boost factor (6) and slop (2) come from configuration - code omitted for simplicity
    PhraseQuery pq = new PhraseQuery() { Boost = tokens.Count * 6, Slop = 2 };
    foreach(var token in tokens)
        pq.Add(new Term(field.Name, token));

    return pq;
```

```
}

IEnumerable<Query> GetIncrementalMatchQuery(IList<string> tokens, FieldDefinition field)
{
    BooleanQuery bq = new BooleanQuery();
    foreach(var token in tokens)
        bq.Add(new TermQuery(new Term(field.Name, token)), Occur.SHOULD);

    //5 comes from config - code omitted
    int upperLimit = Math.Min(tokens.Count, 5);
    for(int match = 2; match <= upperLimit; match++)
    {
        BooleanQuery q = bq.Clone() as BooleanQuery;
        q.Boost = match * 3;
        q.MinimumNumberShouldMatch = match;
        yield return q;
    }
}
```

A query with the following clauses would be produced if "great white shark australia" is supplied as user input and assuming title is the default field and description is the other field against which search is carried out:

- Exact phrase query
    - title:"great white shark australia"~2^24.0
- Combination of incremental boosted boolean queries with minimum number of matches
    - (title:great title:white title:shark title:australia)~2^6.0
    - (title:great title:white title:shark title:australia)~3^9.0
    - (title:great title:white title:shark title:australia)~4^12.0
- Individual term queries
    - title:great
    - description:great
    - title:white
    - description:white
    - title:shark
    - description:shark
    - title:australia
    - description:australia

Create a Custom Analyzer

It is common for our users to use inflected forms of words (e.g. amenities vs amenity, maisonette vs maison) and accented characters. The `StandardAnalyzer` fell short in some areas such as dealing with common inflections, accented characters, plurals and possessive forms of words (e.g. Toronto's mayor - apostrophe). The `EnglishAnalyzer` (https://github.com/apache/lucenenet/blob/master/src/Lucene.Net.Analysis.Common/Analysis/En/EnglishAnalyzer.cs) did a better job as it includes an `EnglishPossessiveFilter` and the `PorterStemFilter` (which removes ing from visiting), but did not deal with accents appropriately.

The solution was to create a custom analyzer based on the EnglishAnalyzer with an added `ASCIIFoldingFilter` to deal with acents. We use a stopfile, hence the overloaded constructor with the `TextReader` parameter.

Enhanced EnglishAnalyzer

```
public class EnhancedEnglishAnalyzer : StopwordAnalyzerBase
{
     public EnhancedEnglishAnalyzer(LuceneVersion matchVersion, CharArraySet stopwords):base(matchVersion, stopwords) {}
     public EnhancedEnglishAnalyzer(LuceneVersion matchVersion, TextReader stopwords):base(matchVersion, LoadStopwordSet(stopw
ords, matchVersion)) {}

    protected override TokenStreamComponents CreateComponents(string fieldName, TextReader reader)
    {
        Tokenizer source = new StandardTokenizer(m_matchVersion, reader);
        TokenStream result = new StandardFilter(m_matchVersion, source);
        result = new EnglishPossessiveFilter(m_matchVersion, result);
        result = new ASCIIFoldingFilter(result);
        result = new LowerCaseFilter(m_matchVersion, result);
        result = new StopFilter(m_matchVersion, result, m_stopwords);
        result = new PorterStemFilter(result);
        return new TokenStreamComponents(source, result);
    }
}
```

The following tokens would be obtained when the sentence My friends are visiting Montréal's engineering institutions is analyzed.

Tokens from different analyzers

StandardAnalyzer	EnglishAnalyzer	EnhancedEnglishAnalyzer

StandardAnalyzer	EnglishAnalyzer	EnhancedEnglishAnalyzer
my	my	my
friends	friend	friend
visiting	visit	visit
montréal's	montréal	montreal
engineering	engin	engin
institutions	institut	institut

Use a StopFile

The default list of stopwords from the EnglishAnalyzer or StandardAnalyzer is quite short. We often needed to add more words to that list - e.g. "my" in the previous example. A very easy way to do this is to use a stopfile.

Give Important Fields a Boost

It is worthwhile to evaluate the relative importance of each field in any given context and as per user's perception. Boosting important fields is necessary to affect the overall search results and their relevance. Whether to go for index-time or query-time boosting boils down to whatever's convenient. In our case though, no boost was applied at index time. Query-time boosting allowed for greater flexibility to test different combination of boost values and removed the need for regenerating the index and redeployments for minor adjustments (by storing boost values in configuration).

Use an NGram filter in Autocomplete Scenarios

In autocomplete scenarios, when suggestions have to be proposed as user types, we found out that using an NGram filter (like EdgeNGram) was better than relying on wildcard or prefix queries. The index size was bigger, but the performance was always better.

©Programagic 2017