Lucene Overview

As Lucene is a 20 year old project, widely known and used. Having never personally used it, I don't think I can really offer anything new or insightful on the topic. I will, instead, simply focus on the features of Lucene I am likely to need to complete my project.

My project is essentially just a search engine over internet browsing history, and the specific things that I need from Lucene are:

- Process new pages/documents as they come in
- index them with their url as the key
- Store/load this index
- Query this index

In this document I'll go through the Lucene documentation [1], picking out things that are likely to be useful. This should all (hopefully) prove rather simple, considering Lucene *is* a search library.

Indexing

Starting from the top, I need to be able to append documents to the index in real time (something I couldn't find any mechanism for within MeTA [2], which I initially thought to use). As the user browses, I want to be able to index the contents of all pages they open. Ideally, I wouldn't need to save the pages to disk before indexing them.

The following snippet of code, from the Lucene documentation [3], shows how I might want to generate an index:

```
Analyzer analyzer = new StandardAnalyzer();

Path indexPath = Files.createTempDirectory("tempIndex");
Directory directory = FSDirectory.open(indexPath);
IndexWriterConfig config = new IndexWriterConfig(analyzer);
IndexWriter iwriter = new IndexWriter(directory, config);
Document doc = new Document();
String text = "This is the text to be indexed.";
doc.add(new Field("fieldname", text, TextField.TYPE_STORED));
iwriter.addDocument(doc);
iwriter.close();
```

Breaking this down:

- `Analyzer`

  > An Analyzer builds TokenStreams, which analyze text. ***It thus represents a policy for extracting index terms from text.*** [4]

  `StandardAnalyzer` appears to be general purpose, but there are also one that specialize in, for example, Japanese text. For this project, `StandardAnalyzer` is likely to be sufficient, though I would like to see at some point if multiple analyzers could be combined or chained together.

- `IndexWriter`

  > An IndexWriter creates and maintains an index. [5]

[1] https://lucene.apache.org/core/9_4_1/index.html
[2] https://meta-toolkit.org/
[3] https://javadoc.io/doc/org.apache.lucene/lucene-core/latest/index.html
[4] https://javadoc.io/static/org.apache.lucene/lucene-core/9.4.1/org/apache/lucene/analysis/Analyzer.html
[5] https://javadoc.io/static/org.apache.lucene/lucene-core/9.4.1/org/apache/lucene/index/IndexWriter.html

The index is saved to disk at the provided directory. Based on the `IndexFiles` demo[6] having an `-update` flag, it seems that it should also be possible to make changes to an existing index, instead of regenerating the entire thing.

- `Document`

  Documents are the unit of indexing and search. [7]

  Before text can be indexed, it needs to be placed within a `Document` object. They do not need to be read in from disk. Additionally, "fieldname" is set explicitly, and isn't some random/automatic id, so it should be trivial to index on URL.

- `TextField.TYPE_STORED`

  A field that is indexed and tokenized, without term vectors. For example this would be used on a 'body' field, that contains the bulk of a document's text. [8]

  It seems it may be possible to index pages *without* storing the contents by using `TextField.TYPE_NOT_STORED`. This may be useful to save on disk usage, if the user doesn't need to be shown context for query results.

So that covers all of my Indexing-related requirements quite nicely, including live updates and storage.

Querying

Pulling more from the example in the Lucene documentation[9], we can see how querying is done as well:

```
// Now search the index:
DirectoryReader ireader = DirectoryReader.open(directory);
IndexSearcher isearcher = new IndexSearcher(ireader);
// Parse a simple query that searches for "text":
QueryParser parser = new QueryParser("fieldname", analyzer);
Query query = parser.parse("text");
ScoreDoc[] hits = isearcher.search(query, 10).scoreDocs;
assertEquals(1, hits.length);
// Iterate through the results:
for (int i = 0; i < hits.length; i++) {
  Document hitDoc = isearcher.doc(hits[i].doc);
  assertEquals("This is the text to be indexed.", hitDoc.get("fieldname"));
}
ireader.close();
directory.close();
IOUtils.rm(indexPath);
```

Let's go over this piece by piece again:

- `IndexSearcher`

  Implements search over a single IndexReader. [...] For performance reasons, if your index is unchanging, you should share a single IndexSearcher instance across multiple searches instead of creating a new one per-search. **If your index has changed and you wish to see the changes reflected in searching, you should use DirectoryReader.openIfChanged(DirectoryReader) to obtain a new reader and then create a new IndexSearcher from that.** [10]

[6] https://lucene.apache.org/core/9_4_1/demo/src-html/org/apache/lucene/demo/IndexFiles.html
[7] https://javadoc.io/doc/org.apache.lucene/lucene-core/latest/org/apache/lucene/document/Document.html
[8] https://lucene.apache.org/core/9_4_1/core/org/apache/lucene/document/TextField.html
[9] https://javadoc.io/doc/org.apache.lucene/lucene-core/latest/index.html
[10] https://javadoc.io/static/org.apache.lucene/lucene-core/9.4.1/org/apache/lucene/search/IndexSearcher.html

This opens an index for querying (in this case a directory). As my index will receive updates pretty regularly (every time the user opens a webpage), some precaution will be necessary to make sure we're querying with those changes accounted for.

- `QueryParser`

  > QueryParser parses the user query string and constructs a Lucene Query object [...] The first parameter to the QueryParser constructor specifies the default search field, which is content field in this case. This default field is used if the query string does not specify the search field. The second parameter specifies the Analyzer to be used when the QueryParser parses the user query string. [11]

- `Query`

  Representation of a user query. We can take the user's query as a string, pass it through the above parser, and receive an object Lucene can use to query the index.

- `ScoreDoc`

  List of documents and their scores. It appears that you can choose the retrieval model used for scoring [12]. This doesn't actually contain the documents themselves, but contains the documents "number" in the index, score, etc. The `10` being pass in here is the number of results we want to get back.

- `hitDoc`

  Gets the actual document referred to in ScoreDoc. Presumably this wouldn't work if we used the `TextField.TYPE_NOT_STORED` mentioned above.

So it also seems that Lucene should cover all of my query related needs. It even appears to have multiple available ranking functions available to try out once I have an index built up.

PyLucene

Having trouble setting up a java environment?

[11] http://web.cs.ucla.edu/classes/winter15/cs144/projects/lucene/index.html
[12] https://lucene.apache.org/core/9_4_1/core/org/apache/lucene/search/package-summary.html