The Apache Foundation's Lucene and Solr Tools

Ian Will

May 2012

Abstract

Lucene is a powerful tool for searching text documents. It is one of the popular tools being used for problems traditionally solved by relational databases. Lucene is developed by the Apache foundation entirely in Java. This report will discusses the basics of Lucene (version 3.6.0) and Solr and presents two sample applications–one that searches the USGS dataset of geographic names (GNIS, `http://geonames.usgs.gov/domestic/index.html`) and another that searches PDF and Microsoft Word documents. It then follows the Solr tutorial, extending it with examples of indexing PDF and Microsoft Word documents and hit-highlighting.

Overview

Lucene is a search library designed index and query text documents and metadata. It is maintained and distributed by the Apache Software Foundation, and is freely available on their website [1]. Lucene has been widely adopted as underlying search technology for web applications and high performance computing. A distributed version of Lucene is available using Apache's clustering library (Hadoop). Lucene was initially developed by Doug Cutting beginning in 1999. It's latest release was version 3.6.0 on April 12, 2012.

Architecture

In Lucene's all searches and indexes revolve around `Document`s and `Field`s. A `Document` contains one or more `Field`s, which may be metadata or text. Each field is a key-value pair–a name and a value. An index of `Document`s is maintained in a `Directory`. `Directory`ies may be a traditional file system directory, or they could be more exotic, such as RAM-only, or memory mapped. The types of directories provided in Lucene 3.6 are described more exhaustively in the following section. Each `Document` consists of a number of `Field`s. A field might be a title, an author, of the text body. Each field is indexed and queried individually.

`Document`s are added to a `Directory` by an `IndexWriter`. Conversely, at query time a `Directory` is opened by an `IndexReader`, searched using an `IndexSearcher` and a `Query`, and scores are collected sorted and accessed by a `Collector`. Yonik Seeley provides a good overview in his presentation, from which figure 1 is borrowed.

When adding a `Document` to an index, one must specify a few options for each field which control how that field can be accessed later. A field may be stored in the index, which means its full value will be available in the query results. If a field is not stored, the index saves some space, but the field value must be manually looked up as part of the query processing. Processing an unstored field requires extra work in the same way processing a foreign key would require an extra JOIN or table lookup in a relational database. A field may or may not be indexed, and if it is not its value will not be searched (unlike relational databases, the index is the only way Lucene performs searches).

Combined with these two options, a field may or may not be analyzed. An analyzed field is broken up into tokens (words) by an `Analyzer`, eliminating whitespace and common words, and possibly stemming a number of suffixes to generate matches on a range of similar words.

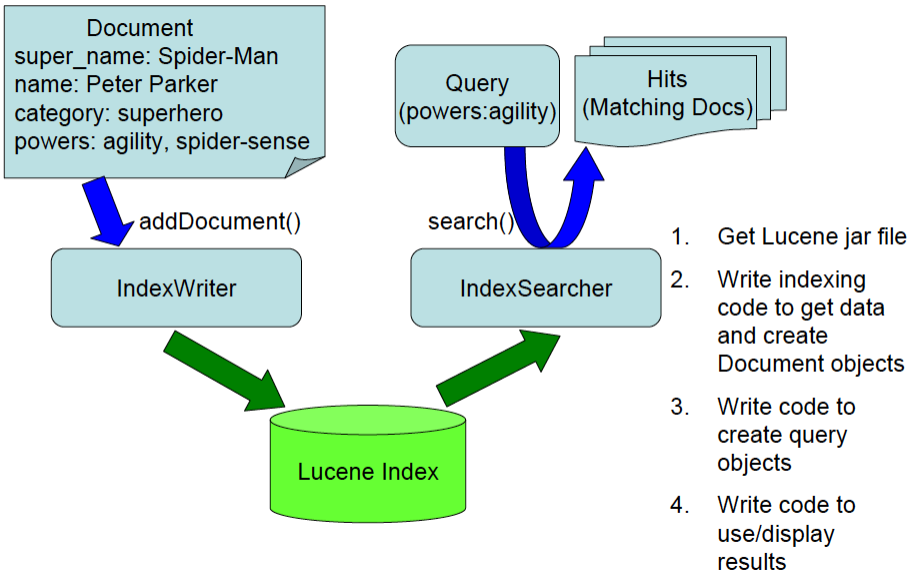

Figure 1: Overview of Lucene Architecture [14]

Directory Implementations

Lucene stores an index in a `Directory` object. `Documents` with `Fields` are added to this `Directory` by an `IndexWriter`. Later, an `IndexReader` opens the `Directory` index and uses an `IndexSearcher` to search for a `Query`. Results are returned to a score Collector, which groups and provides access to the Documents. This is the basic workflow used with Lucene.

In your first Lucene program, you may use a RAMDirectory to store a small set of documents and indexes in RAM. Later you'll have to choose which Directory implementation to choose from. The options are

1. `MMapDirectory`: maps an entire file into a portion of the JRE's address space. Care should be taken to insure the file can fit inside the address space, thus 64 bit JRE's are recommended with this Directory.

2. `NIOFSDirectory`: file system directory backed by java.nio FileChannels, allowing multi-threaded reading without synchronizing. Windows users are cautioned about a bug in the Windows implementation of FileChannel.read ()

3. `SimpleFSDirectory`: file system directory backed by java.io.RandomAccessFile, poor concurrent performance (NIOFS or MMap are recommended)

4. `NRTCachingDirectory`: designed for near-real-time searches, wraps a delegate Directory with a RAMDirectory for faster searches. Handles merging data between them as necessary.

5. `RAMDirectory`: memory-resident index, should not be used for more than a few hundred megabytes of data

6. `FileSwitchDirectory`: partitions files into two sub Directory instances based on file extension, resulting in one primary Directory and a secondary Directory

7. `CompoundFileReader`: (package protected in org.apache.lucene.index) read-only Directory wrapper for compound streams.

Analyzer Implementations

An `Analyzer`'s main job is to provide a `TokenStream` via `Analyzer.tokenStream(String, Reader)`. A `TokenStream`'s main method is `TokenStream.incrementToken()`; it steps through a stream of tokens one at a time. There are two major branches of the `TokenStream` hierarchy–`TokenFilter`, which takes as input another `TokenStream`; and `Tokenizer`, which takes input from a `java.io.Reader`.

There are many provided implementations of `Tokenizer` and `TokenStream`. There are also a few stock implementations of `Analyzer`, which return common chainings of `TokenFilter`s.

The provided `Tokenizer` implementations include the following.

1. `LetterTokenizer`: Defines tokens as maximal strings of adjacent letters, using `java.lang.Character.isLetter()` to identify characters. This does a poor job for Asian languages, which are not separated by spaces.
2. `LowerCaseTokenizer`: Extends `LetterTokenizer`, normalizing all tokens to lower case.
3. `WhitespaceTokenizer`: Divides text at whitespace, forming tokens comprised of adjacent sequence of non-Whitespace characters.
4. `ClassicTokenizer`: Divides text at punctuation characters (exclusively). Has some additional smarts to recognize email address and similar common exceptions as single tokens. Uses JFlex (`http://jflex.de/`), an open source lexical analyzer generate (scanner generator).
5. `KeywordTokenizer`: Forms tokens comprised of the entire available contents of the underlying `Reader`.
6. `StandardTokenizer`: Divides text into tokens based on Word Break rules from the Unicode Text Segmentation algorithm specified in Unicode Standard Annex #29 (`http://unicode.org/reports/tr29/`). `StandardTokenizer` also uses JFlex for lexical analysis.
7. `UAX29URLEmailTokenizer`: Divides text into tokens based on the Unicode Text Segmentation algorithm like `StandardTokenizer`, but also tokenizes URLs and email addressed according to relevant RFCs.

There are many provided implementations of `TokenFilter`, the most interesting of which are listed below. Each of these implementations overrides the `TokenStream.incrementToken()` method, effectively blocking out contents of the stream as it is being accessed.

1. `ASCIIFoldingFilter`: Converts characters outside the 8-bit ASCII range (e.g. from the Basic Latin Unicode block) into 8-bit ASCII equivalents. Only characters with reasonable ASCII equivalents are converted.
2. `CachingTokenFilter`: Caches stream tokens for efficient future access. (Specifically, these are `AttributeSource.State` objects).
3. `ClassicFilter`: Removes apostrophes from the ends of word tokens and dots from acronyms.
4. `CollationKeyFilter`: Uses `java.text.Collator` to convert each token into a `java.text.CollationKey`
5. `LengthFilter`: Removes tokens that are too long or too short
6. `StopFilter`: Filters out stop words
7. `ISOLatin1AccentFilter`: (deprecated, use `ASCIIFoldingFilter`)
8. `KeywordMarkerFilter`: Takes a set of strings designating keywords and sets `CharTermAttribute.setKeyword(true)` for tokens contained in the supplied set.
9. `LowerCaseFilter`: Normalizes all tokens to lower case.

FEATURE_ID	FEATURE_NAME	FEATURE_CLASS	STATE_ALPHA
STATE_NUMERIC	COUNTY_NAME	COUNTY_NUMERIC	PRIMARY_LAT_DMS
PRIM_LONG_DMS	PRIM_LAT_DEC	PRIM_LONG_DEC	SOURCE_LAT_DMS
SOURCE_LONG_DMS	SOURCE_LAT_DEC	SOURCE_LONG_DEC	ELEV_IN_M
ELEV_IN_FT	MAP_NAME	DATE_CREATED	DATE_EDITED

Table 1: Field names available in the USGS National File

10. `PorterStemFilter`: Uses the `PorterStemmer` class to stem all non-keyword tokens. This requires that all tokens already be in lower case. Thus you will need to establish a chain of `TokenFilter`s, e.g. `new PorterStemFilter(new LowerCaseTokenizer( new KeywordMarkerFilter( reader )))`. This uses the stemming algorithm described by Porter in [11] to condense tokens into root words by removing spurious suffixes.

11. `StandardFilter`: This implementation appears to be identical to `ClassicFilter`; it strips apostrophes and acronym dots from tokens.

12. `TeeSinkTokenFilter`: Allows multiple streams to share common attribute states.

Querying

There are two ways to interact with the lucene query system: via the built in `QueryParser`, or directly via the API (e.g. by adding `TermFilter`s) [5]. The `QueryParser` is designed to parse human search entries, whereas the API is recommended for programmatically generated queries.

This paper will first discuss Lucene's processing of human entered queries, and will then follow with a comparison to equivalent operations using the Java API. As query terms are discussed, it's useful to have an example in mind. This paper will use a Lucene index built from the USGS National File of places [16]. The available field names in the USGS dataset are shown in table . The example application can be run unzipping the examples archive and running:

```
java -jar examples.jar edu.gwu.spatial.SpatialUI
```

Type Lucene search queries into the bar and results will be displayed in the list and returned on the map as you type. Results will be restricted to the current bounds of the map, but movements to the map view don't trigger searches. The interface is shown in 2. To begin, use the "Index file..." button to select a version of the USGS NationalFile (state files should also work). It takes a few seconds to build the index, status will be displayed in the console. After the index has loaded, type in the search bar to begin searching.

When instantiated, the query parser takes a default field. All strings entered will search the default field. To search alternate files, prefix the field name followed by a colon for example (STATE_NAME:WV). Terms can be modified by wildcards (? and *) matching a single character or multiple characters respectively.

Fuzzy search can be invoked by appending a tilde (~) to the end of a term, e.g. (Cooran~AND STATE_ALPHA:DC) will find a surprising number of Corcoran related matches in DC. This fuzzy search term can also be modified by a similarity parameter between 0 and 1 (e.g. Cooran~0.8), which only returns matches above that threshold of similarity. Lucene's fuzzy searches use the Levenshtein distance algorithm to quantify closeness.

Using the tilde operator on a group of terms restricts matches to those in which both terms appear within some distance of each other, for example "Corcoran statue" 1 finds the Corcoran Lions Statue in DC, when "Corcoran statue" would not.

Ranges of values can be searched using square brackets [ ] or curly braces { } for inclusive and exclusive bounds respectively. Do search all states beginning with D, you could use "STATE_ALPHA:[DA TO DZ]".

Relative importance of terms can be modified using the boost operator ^ followed by a boost multiplier (must be positive, but can be less than 1). By default, all terms have boost of 1. A boost values less than

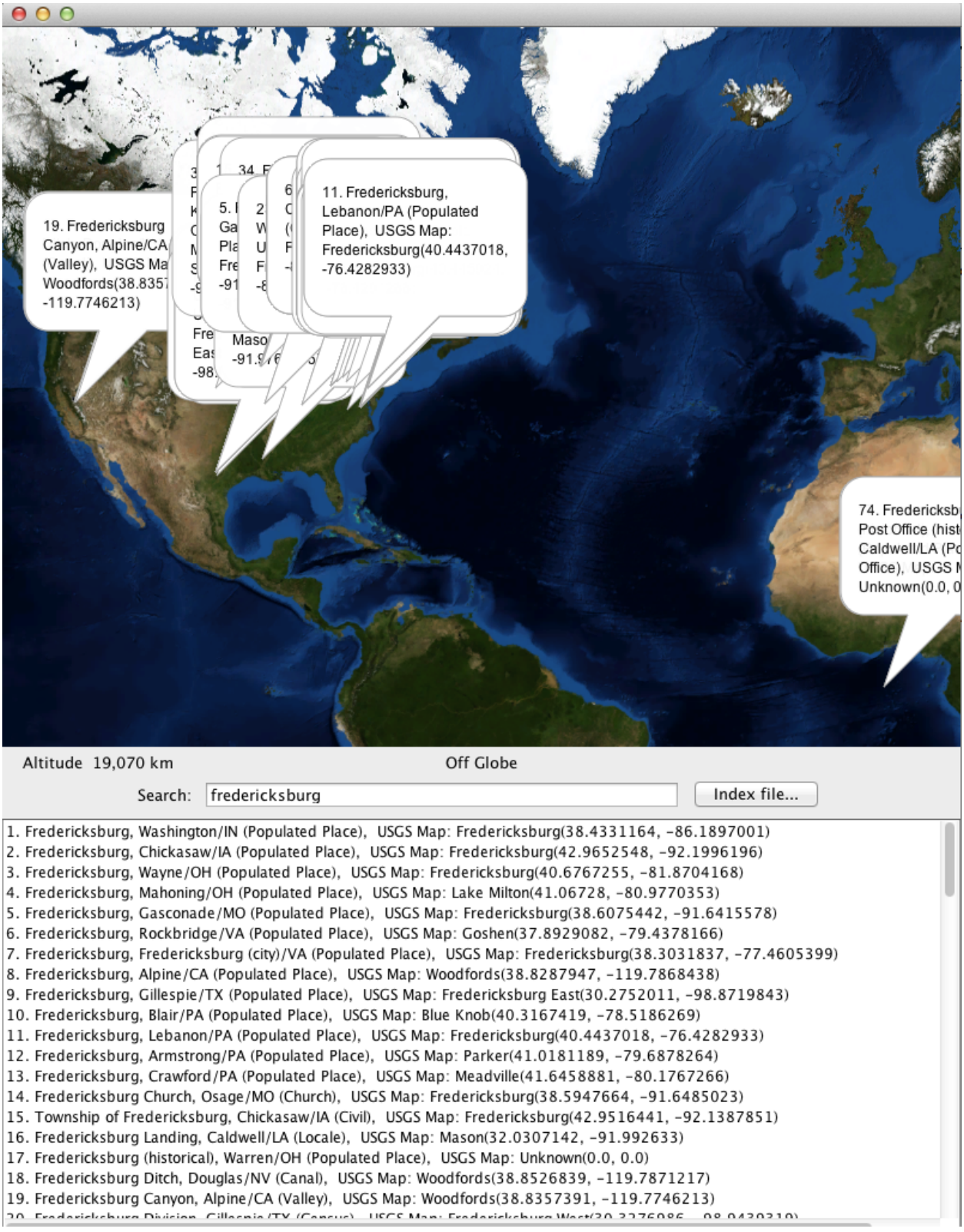

Figure 2: User interface for spatial search example

Query Implementation	Purpose	Sample expressions
`TermQuery`	Single term query, which effectively is a single word.	reynolds
`PhraseQuery`	A match of several terms in order, or in near vicinity to one another.	"light up ahead"
`RangeQuery`	Matches documents with terms between beginning and ending terms, including or excluding the end points.	[A TO Z], {A TO Z}
`WildcardQuery`	Lightweight, regular-expression-like term-matching syntax.	j*v?, f??bar
`PrefixQuery`	Matches all terms that begin with a specified string.	cheese*
`FuzzyQuery`	Levenshtein algorithm for closeness matching.	tree~
`BooleanQuery`	Aggregates other Query instances into complex expressions allowing AND, OR, and NOT logic.	reynolds AND "light up ahead", cheese* -cheesewhiz

Table 2: Lucene query syntax and associated `Query` classes

1 will decrease its boost relative to other terms, while a value greater than 1 will make that term more important.

Boolean operators are supported for query terms. These include AND, OR, NOT (require capitalization), "+", and "-". The OR term is the default conjunction, so if two terms are combined without specifying another operator, OR is used. The "+" operator precedes a required term (e.g. "+Corcoran"), and the "-" operator precedes a forbidden term (e.g. "Corcoran -statue"). Terms can also be grouped together using parenthesis, which can combine an arbitrary number of terms and binary expressions (e.g. "(Corcoran -statue) AND STATE_ALPHA:DC").

Specifying queries via the API uses the same basic ideas previously discussed, except each is accomplished by using and combining implementations of the `Query` class. Erik Hatcher has a good article comparing how human entered query strings are parsed by `QueryParser` and the equivalent `Query` implementations in the API [7]. Table is reproduced as a reference from the article.

Searching for Numeric Values

Lucene is designed to index text, and all indexes thus natively index text-like fields. There are no native numeric types as with typical databases [13]. To do queries on numerical values and ranges, the values must be converted to strings for indexing and the search terms must be converted using the same process. Different techniques are used depending on the type of number and desired applications.

One popular data structure for storing numeric ranges as string while retaining ordering (e.g. where $1 < 2$ and $11 > 2$; in the typical lexical ordering 11 is less than 2) is called a Trie [9]. A Trie stores tokens in a tree, where each node represents a single character and the leaf represents the full token. Each successive tree level inherits the prefix characters from its parents. A string representation of a number can then be inserted into a trie traversed in numeric order.

Another popular data structure for representing geographic positions is a Geohash, which maps a latitude and longitude to a short string of characters. Pairs of coordinates are grouped into grid-shaped buckets, so that coordinates that are near each other share the same prefix. This is useful for database searches, because two fields (latitude and longitude) are grouped into one.

Recent versions of Lucene include built in classes to help convert numeric values into strings for indexing, so the conversion from number to string is mostly transparent to the programmer. To treat certain fields as numeric, simple add a `NumericField` to a `Document` when building the index. Examples of building an index and querying numeric fields are shown in listings 1 and 2. Lucene uses a trie data structure to build the index of `NumericField`s internally.

Listing 1: Creating numeric fields in Lucene

```
IndexWriterConfig config = new IndexWriterConfig(Version.LUCENE_36, analyzer);
IndexWriter w = new IndexWriter(index, config);
Document doc = new Document()
NumericField num = new NumericField(curHeader, Field.Store.YES, true);
num.setDoubleValue(Double.parseDouble(curData));
doc.add(num);
w.addDocument(doc);
w.close();
```

Listing 2: Querying numeric fields in Lucene

```
NumericRangeQuery<Double> lonRange = NumericRangeQuery.newDoubleRange(
    "PRIM_LONG_DEC",
    nw.getLongitude().getDegrees(),
    se.getLongitude().getDegrees(),
    true, true);
int hitsPerPage = 100;
IndexReader reader = IndexReader.open(index);
IndexSearcher searcher = new IndexSearcher(reader);
TopScoreDocCollector collector = TopScoreDocCollector.create(hitsPerPage, true);

searcher.search(lonRange, collector);
ScoreDoc[] hits = collector.topDocs().scoreDocs;
```

Note in listing 1 the use of the expanded constructor, passing in `Field.Store.YES` and `true` for indexing. By default `NumericField`s are not stored, which means they will not be present in in the returned `ScoreDoc`s. This is useful for query filtering if the numeric data isn't need in results processing. In the sample application, the coordinates of each hit are necessary to display in formation on the globe, thus they need to be stored for access at query time.

Using the USGS domestic names dataset (`http://geonames.usgs.gov/docs/stategaz/NationalFile_20120416.zip`) [16] with the above queries results in search times of between 200 and 300 ms. Using the graphical search tool, a query for fredericksburg at the full zoom level returned 100 results (the maximum number) in 299 ms. Zooming in on a Fredericksburg, VA returned 49 results in about the same time (302 to 226 ms).

Querying and Filtering

With the example above, the numeric range was restricted using a NumericRangeQuery. Similar effect can also be accomplished by using a Filter with the initial Query as part of a FilteredQuery. The example above would look similar to listing 3. Filtering seems to exhibit poorer performance than range querying. Searches for fredericksburg using a range filter took between 289 and 966 ms.

Listing 3: Filtering on numeric fields in Lucene

```
Query q = new QueryParser(Version.LUCENE_36, "FEATURE_NAME", analyzer).parse(query);
NumericRangeFilter<Double> latRange = NumericRangeFilter.newDoubleRange(
        "PRIM_LAT_DEC",
        se.getLatitude().getDegrees(),
        nw.getLatitude().getDegrees(),
        true, true);
//west will be lower than east....
NumericRangeFilter<Double> lonRange = NumericRangeFilter.newDoubleRange(
        "PRIM_LONG_DEC",
        nw.getLongitude().getDegrees(),
```

```
        se.getLongitude().getDegrees(),
        true, true);

int hitsPerPage = 100;
IndexReader reader = IndexReader.open(index);
IndexSearcher searcher = new IndexSearcher(reader);
TopScoreDocCollector collector = TopScoreDocCollector.create(hitsPerPage, true);

FilteredQuery fq = new FilteredQuery(new FilteredQuery(q,latRange), lonRange);

searcher.search(fq, collector);
ScoreDoc[] hits = collector.topDocs().scoreDocs;
```

This completes the section on numeric and spatial querying with Lucene. The example code can be found in the `edu.gwu.spatial` package.

Indexing PDF Documents

Java does not have native support for reading PDFs, however Apache provides a library for reading PDFs called PDFBox [2]. PDFBox can read PDF documents and associated PDF metadata. There is a convenient way to use PDFBox with Lucene, which is not included in the default jar distribution of PDFBox (to prevent PDFBox from being dependent on Lucene) [10]. It is maintained in the PDFBox source repository and can be built if needed. For the text search example program discussed here, LucenePDFDocument was built and packaged into lucene-pdf.jar. Using the `LucenePDFDocument` class, adding a PDF file to the lucene index is as simple as listing 4. Note that LucenePDFDocument requires apache commons logging and fontbox libraries on the classpath at runtime.

Listing 4: Adding a pdf to a Lucene index

```
public void addFile(File file) throws IOException
{
    IndexWriterConfig config = new IndexWriterConfig(Version.LUCENE_36, analyzer);
    IndexWriter w = new IndexWriter(index, config);

    if(file.getName().toLowerCase().endsWith("pdf"))
    {
        Document pdfDoc = LucenePDFDocument.getDocument(file);
        w.addDocument(pdfDoc);
    }
    w.close();
}
```

This adds all extractable files to a Lucene `Document`, ready to be added to the index. Listing 5 shows the field names and values placed in the Document when run on a draft version of this pdf.

Listing 5: Field names and values as extracted by LucenePDFDocument

```
path: /Users/will/Documents/Courses/databases/final/lucene-report.pdf
url: /Users/will/Documents/Courses/databases/final/lucene-report.pdf
modified: 20120514143419
uid: Users will Documents Courses databases final lucene-report.pdf 20120514143419
contents: null
CreationDate: 20120514143418
Creator: TeX
ModificationDate: 20120514143418
Producer: pdfTeX-1.40.12
Trapped: False
summary: The Apache Foundations Lucene and Solr Tools
```

```
Ian Will
May 2012
Abstract
Lucene is a powerful tool for searching text documents. It is one of the popular tools being
     used for problems traditionally solved by relational databases. Lucene is developed by
    the Apache foundation entirely in Java. This report will discusses the basics of Lucene
    (version 3.6.0) and Solr and presents two sample applications     one that searches the
    USGS dataset of geographic names (GNIS, http://geonames.usgs.gov/domesti
```

The metadata fields are properly extracted from the pdf, as is the summary. Notice that the contents field is null. This is because the contents are indexed, but not stored. Thus searches can be conducted on the full contents of the text document, but only a 500 character summary is stored. The path and url to the full file are stored, so a search can quickly access the full text if necessary. The summary is stored, but not indexed. Thus the summary field can't be searched directly. But this is not necessary, because the full text (including the summary) is indexed under the contents field.

Hit Highlighting

Lucene has the ability to highlight the tokens that contributed to a document's high search score. This is done with the highlighter package in Lucene contrib. The necessary jars can be found in the contrib/highlighter directory of a Lucene binary release. To use hit highlighting, a field must be stored in the index. Therefore, to implement hit highlighting with our above example we'll need to store the contents field with the index. We can do this by extracting the text using PDFBox and adding another "contents" field with `Store.YES`. See listing 6.

Listing 6: Storing contents field in index

```
public void addFilePDFBox(File file) throws IOException
    {
        IndexWriterConfig config = new IndexWriterConfig(Version.LUCENE_36, analyzer);
        IndexWriter w = new IndexWriter(index, config);

        if(file.getName().toLowerCase().endsWith("pdf"))
        {

            Document pdfDoc = LucenePDFDocument.getDocument(file);
            List<Fieldable> fields = pdfDoc.getFields();
            for(Fieldable field : fields)
            {
                System.out.println("Field name: "+field.name()+", field value: "+field.stringVal
            }

            //Extract the text body and add to Document with Store.YES
            PDFParser parser = new PDFParser(new FileInputStream(file));
            parser.parse();
            COSDocument cd = parser.getDocument();
            PDFTextStripper stripper = new PDFTextStripper();
            String text = stripper.getText(new PDDocument(cd));
            pdfDoc.add(new Field("contents", text, Store.YES, Index.ANALYZED));
            w.addDocument(pdfDoc);
            cd.close();
        }
        w.close();
    }
```

Once the contents field is stored in the index, matching terms can be highlighted as shown in listing 7. An example of the resulting matches is shown in figure 4.

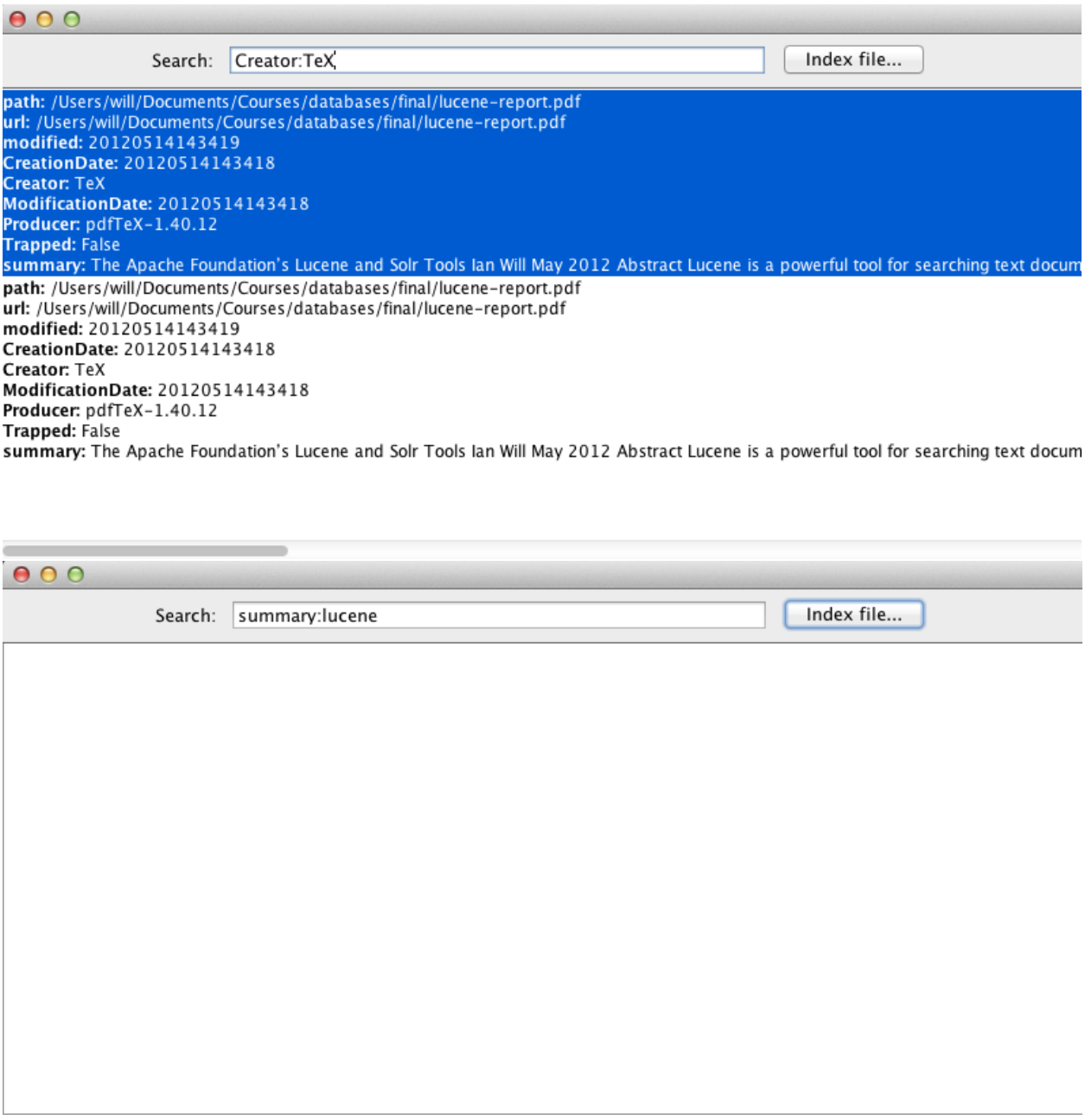

Figure 3: Searching the Creator field and the summary field

Listing 7: Highlighting search terms

```
private String highlightResults(Query q, Fieldable f) throws IOException, InvalidTokenOffsetsExc
{
    TokenStream ts = TokenSources.getTokenStream(
        "contents",
        f.stringValue(),
        analyzer);
    QueryScorer scorer = new QueryScorer(q, "contents");
    scorer.init(new CachingTokenFilter(ts));

    Fragmenter frag = new SimpleSpanFragmenter(scorer, 30);
    Highlighter highlighter = new Highlighter(scorer);
    highlighter.setTextFragmenter(frag);
    String[] fragments = highlighter.getBestFragments(
        ts,
        f.stringValue(),
        5);
    StringBuffer res = new StringBuffer();
    res.append("<b>Best matches:</b><br/>");
    for(String fragment : fragments)
    {
        res.append(fragment);
        res.append("<br/>");
    }
    return res.toString();
}
```

Listing 8: Query processing

```
public List findResults(String query) throws ParseException, CorruptIndexException, IOException,
    {
        Query q = new QueryParser(
            Version.LUCENE_36,
            "contents",
            analyzer).parse(query);

        int hitsPerPage = 100;
        IndexReader reader = IndexReader.open(index);
        IndexSearcher searcher = new IndexSearcher(reader);
        TopScoreDocCollector collector = TopScoreDocCollector.create(
            hitsPerPage, true);

        try
        {
            searcher.search(q, collector);

            ScoreDoc[] hits = collector.topDocs().scoreDocs;
            List results = new ArrayList(hits.length);
            for(int i=0; i<hits.length; i++)
            {
                Document d = searcher.doc(hits[i].doc);
                StringBuffer res = new StringBuffer();
                res.append("<html>");
                Fieldable f = d.getFieldable("path");
                if(f != null)
                {
```

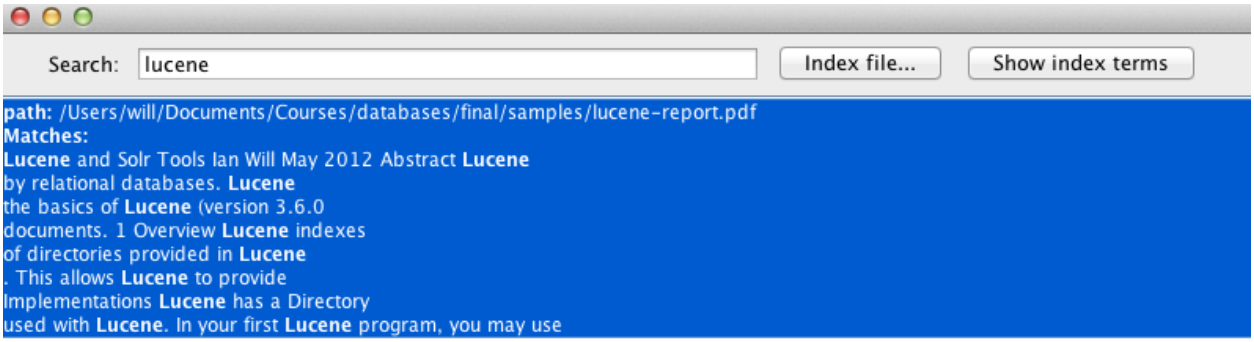

Figure 4: Example of hit highlighting

```
                res.append("<b>");
                res.append(f.name()+": ");
                res.append("</b>");
                res.append(f.stringValue());
                res.append("<br/>");
            }
            f = d.getFieldable("contents");
            if(f != null)
            {
                res.append(highlightResults(q, f));
            }
            res.append("</html>");
            results.add(res.toString());
        }

        return results;
    }finally
    {
        searcher.close();
    }
}
```

Indexing Microsoft Word and PDF using Apache Tika

In addition to the PDFBox [2] library, Apache also supports a package called Tika [4], which which packages both PDFBox and Apache POI (which supports Microsoft document formats) under a common interface. Using Tika to extract metadata from both PDF and Microsoft Word documents is shown in listing 9. The code is based on an example by John Reese [12]. This example stores the contents field in the index, making it compatible with the hit highlighting example discussed previously.

Listing 9: Parsing PDF and Microsoft Word metadata with Apache Tika

```
public void addFileTika(File file) throws IOException, SAXException, TikaException
{
    IndexWriterConfig config = new IndexWriterConfig(Version.LUCENE_36, analyzer);
    IndexWriter w = new IndexWriter(index, config);

    FileInputStream is = new FileInputStream(file);

    BodyContentHandler contenthandler = new BodyContentHandler();
    Metadata metadata = new Metadata();
    metadata.set(Metadata.RESOURCE_NAME_KEY, file.getName());
    Parser parser = new AutoDetectParser();
    ParseContext ctxt = new ParseContext();
    parser.parse(is, contenthandler, metadata, ctxt);

    Document doc = new Document();
    doc.add(new Field(
        "name",
        file.getName(),
        Field.Store.YES,
        Field.Index.NOT_ANALYZED));
    doc.add(new Field(
        "path",
        file.getCanonicalPath(),
        Field.Store.YES,
        Field.Index.NOT_ANALYZED));
    if(metadata.get(Metadata.TITLE) != null)
    {
        doc.add(new Field(
            "title",
            metadata.get(Metadata.TITLE),
            Field.Store.YES,
            Field.Index.ANALYZED));
    }
    if(metadata.get(Metadata.AUTHOR) != null)
    {
        doc.add(new Field(
            "author",
            metadata.get(Metadata.AUTHOR),
            Field.Store.YES,
            Field.Index.ANALYZED));
    }
    String contents = contenthandler.toString();
    System.out.println("Contents: "+contents.length());
    doc.add(new Field("contents",contents,Field.Store.YES,Field.Index.ANALYZED));

    w.addDocument(doc);
    w.close();
}
}
```

This completes the example of indexing PDF and Microsoft Word documents using Apache Lucene. The example code can be found in the gwu.edu.text package, with `LuceneTextIndex` interfacing with Apache Lucene classes, and TextUI providing a Swing based front-end. To run the example, call `java -jar lucene-example.jar gwu.edu.text.TextUI`. Use the "Index file..." button to add PDF or Microsoft Word documents to the index. Type in the search bar to enter a search query in Lucene syntax (as discussed above). The search will be done as you type, and results updated

in the list below. The “Show index terms” button replaces the search results list with a list of all the terms currently in the Lucene index.

Solr

Solr [3] is a full-text search server built on Lucene which runs in a servlet container (e.g. Tomcat). It is an HTTP wrapper around Lucene, adding with extended search terms and automatic inclusion of some of the non-standard Lucene contrib libraries (like Tika). Many of the extension (contrib) features of Lucene like Tika, hit highlighting, and spatial search are included in a standard Solr distribution.

Solr provides an easy to run example, and a tutorial [15] which walks through the basic steps of starting a server, posting, searching, and deleting documents from the index. The examples are all in the example directory of a standard Solr download. The example setup processing includes starting the servlet in Jetty (`java -jar start.jar`), posting sample XML documents `java -jar post.jar solr.xml`, and querying results. The post.jar archive contains a single class–`SimplePostTool`, which is designed specifically to work with the tutorial. It opens an HTTP connection to a specified URL (if unspecified, it uses the recommended tutorial ULR of “http://localhost:8983/solr/update”) and posts the contents of the specified files.

Listing 10: Solr XML input

```
<add>
<doc>
  <field name="id">SOLR1000</field>
  <field name="name">Solr, the Enterprise Search Server</field>
  <field name="manu">Apache Software Foundation</field>
  <field name="cat">software</field>
  <field name="cat">search</field>
  <field name="features">Advanced Full-Text Search Capabilities using Lucene</field>
  <field name="features">Optimized for High Volume Web Traffic</field>
  <field name="features">Standards Based Open Interfaces - XML and HTTP</field>
  <field name="features">Comprehensive HTML Administration Interfaces</field>
  <field name="features">Scalability - Efficient Replication to other Solr Search Servers</field>
  <field name="features">Flexible and Adaptable with XML configuration and Schema</field>
  <field name="features">Good unicode support: h&#xE9;llo (hello with an accent over the e)</field>
  <field name="price">0</field>
  <field name="popularity">10</field>
  <field name="inStock">true</field>
  <field name="incubationdate_dt">2006-01-17T00:00:00.000Z</field>
</doc>
</add>
```

The example solr.xml is shown in listing 10. The Solr xml file defines the file name and values for a given document. This is the same structure of Document and Field that Lucene uses.

Solr can be queried using the select URL. For example

```
wget -O - http://localhost:8983/solr/select?q=solr
```

returns the XML in listing 11. Each field is group into an `<arr>` tag, with each value listed as a string element.

Listing 11: XML response to select/?q=solr

```
 <?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
  <int name="status">0</int>
  <int name="QTime">0</int>
  <lst name="params">
  <str name="q">solr</str>
</lst></lst>
<result name="response" numFound="1" start="0">
<doc>
    <arr name="cat">
        <str>software</str>
        <str>search</str>
```

```
    </arr>
    <arr name="features">
        <str>Advanced Full-Text Search Capabilities using Lucene</str>
        <str>Optimized for High Volume Web Traffic</str>
        <str>Standards Based Open Interfaces - XML and HTTP</str>
        <str>Comprehensive HTML Administration Interfaces</str>
        <str>Scalability - Efficient Replication to other Solr Search Servers</str>
        <str>Flexible and Adaptable with XML configuration and Schema</str>
        <str>Good unicode support: h llo (hello with an accent over the e)</str>
    </arr>
    <str name="id">SOLR1000</str>
    <bool name="inStock">true</bool>
    <date name="incubationdate_dt">2006-01-17T00:00:00Z</date>
    <str name="manu">Apache Software Foundation</str>
    <str name="name">Solr, the Enterprise Search Server</str>
    <int name="popularity">10</int><float name="price">0.0</float>
    <str name="price_c">0,USD</str>
</doc>
</result>
</response>
```

Solr PDF and Microsoft Word Support

In Solr, Apache's Tika library is used to provide support for PDF and Microsoft Word documents with no extra configuration required. An example of posting a PDF or MS Word document to Solr using curl is shown in listing 12.

Listing 12: Using curl to add a pdf document to Solr

```
curl "http://localhost:8983/solr/update/extract?literal.id=doc1&commit=true" -F "
    myfile=@lucene-report.pdf"
```

Then a simple search for the term "lucene" results in the XML response shown in listing 13.

Listing 13: Response to search?q=lucene

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">2</int>
    <lst name="params">
        <str name="q">lucene</str>
    </lst>
</lst>
<result name="response" numFound="2" start="0">
<doc>
    <arr name="cat">
        <str>software</str>
        <str>search</str>
    </arr>
    <arr name="features">
        <str>Advanced Full-Text Search Capabilities using Lucene</str>
        <str>Optimized for High Volume Web Traffic</str>
        <str>Standards Based Open Interfaces - XML and HTTP</str>
        <str>Comprehensive HTML Administration Interfaces</str>
        <str>Scalability - Efficient Replication to other Solr Search Servers</str>
        <str>Flexible and Adaptable with XML configuration and Schema</str>
        <str>Good unicode support: h llo (hello with an accent over the e)</str>
    </arr>
    <str name="id">SOLR1000</str>
    <bool name="inStock">true</bool>
    <date name="incubationdate_dt">2006-01-17T00:00:00Z</date>
    <str name="manu">Apache Software Foundation</str>
```

```
    <str name="name">Solr, the Enterprise Search Server</str>
    <int name="popularity">10</int>
    <float name="price">0.0</float>
    <str name="price_c">0,USD</str>
    </doc>
<doc>
    <arr name="content_type">
        <str>application/pdf</str>
    </arr>
    <str name="id">doc1</str>
    <date name="last_modified">2012-05-14T15:32:49Z</date>
</doc>
</result>
</response>
```

Note that two documents were found, one is from the Solr tutorial example file solr.xml, the other is the PDF posted previously. Notice that the PDF has no metadata, summary, or content. It only has its unique document id which we specified in the curl command as "doc1." This is because the Tika "content" field (which we are familiar with from the Lucene Tika example above) is mapped by Solr to a field called "text" which is indexed but not stored (as we saw in `LucenePDFDocument`). This mapping is defined in the `/update/extract` handler in `solrconfig.xml`, and can be overridden in the URL as well.

To override the default mapping of "content" to "text" in the URL, change the curl line as shown in listing (as described in [6]).

```
curl "http://localhost:8983/solr/update/extract?literal.id=doc1&uprefix=attr_&fmap
    .content=attr_content&commit=true" -F "myfile=@lucene-report.pdf"
```

Two terms have been added to the URL: `uprefix=attr_` and `fmap.content=attr_content`. The first causes unrecognized fields to be given the attr_ prefix, which causes them to be stored. The second URL term overrides the default mapping from "content" to "text," instead mapping it to the "attr_content" field in Solr. Now the contents of the attr_content field can be explicitly searched as shown in listing 14.

```
wget -O - http://localhost:8983/solr/select?q=attr_content:lucene
```

Listing 14: Results of searching attr_content

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
<int name="status">0</int>
<int name="QTime">0</int>
<lst name="params">
    <str name="q">attr_content:lucene</str>
</lst></lst>
<result name="response" numFound="1" start="0">
<doc>
<arr name="attr_content">
<str>The Apache Foundations Lucene and Solr Tools
  Ian Will
  May 2012
  Abstract
  Lucene is a powerful tool for searching text documents. It is one of the popular tools
      being used for problems traditionally solved by relational databases. Lucene is
      developed by the Apache foundation entirely in Java. This report will discusses the
      basics of Lucene (version 3.6.0) and Solr and presents two sample applications--one
      that searches the USGS dataset of geographic names
...
</str>
</arr>
<arr name="attr_created">
    <str>Mon May 14 11:32:49 EDT 2012</str>
</arr>
<arr name="attr_creation_date">
    <str>2012-05-14T15:32:49Z</str>
```

```
</arr>
<arr name="attr_creator">
    <str>TeX</str></arr>
<arr name="attr_meta">
    <str>stream_source_info</str>
    <str>myfile</str>
    <str>Last-Modified</str>
    <str>2012-05-14T15:32:49Z</str>
    <str>creator</str>
    <str>TeX</str>
    <str>xmpTPg:NPages</str>
    <str>9</str>
    <str>Creation-Date</str>
    <str>2012-05-14T15:32:49Z</str>
    <str>trapped</str>
    <str>False</str>
    <str>stream_content_type</str>
    <str>application/octet-stream</str>
    <str>created</str>
    <str>Mon May 14 11:32:49 EDT 2012</str>
    <str>stream_size</str>
    <str>228117</str>
    <str>stream_name</str>
    <str>lucene-report.pdf</str>
    <str>producer</str>
    <str>pdfTeX-1.40.12</str>
    <str>Content-Type</str>
    <str>application/pdf</str>
    <str>PTEX.Fullbanner</str>
    <str>This is pdfTeX, Version 3.1415926-2.3-1.40.12 (TeX Live 2011) kpathsea version
        6.0.1</str>
</arr>
<arr name="attr_producer">
    <str>pdfTeX-1.40.12</str>
</arr>
<arr name="attr_ptex_fullbanner">
    <str>This is pdfTeX, Version 3.1415926-2.3-1.40.12 (TeX Live 2011) kpathsea version
        6.0.1</str>
</arr>
<arr name="attr_stream_content_type">
    <str>application/octet-stream</str>
</arr>
<arr name="attr_stream_name">
    <str>lucene-report.pdf</str>
</arr>
<arr name="attr_stream_size">
    <str>228117</str>
</arr>
<arr name="attr_stream_source_info">
    <str>myfile</str>
</arr>
<arr name="attr_trapped">
    <str>False</str>
</arr>
<arr name="attr_xmptpg_npages">
    <str>9</str>
</arr>
<arr name="content_type">
    <str>application/pdf</str>
</arr>
<str name="id">doc1</str>
<date name="last_modified">2012-05-14T15:32:49Z</date>
</doc>
</result>
</response>
```

Listing 14 should look similar to figure 3; the same metadata tags are present though they are a good deal more verbose in Solr's XML. An alternative to overriding the field names in the URL is to edit Solr's `schema.xml` file to define text fields to be stored. To do this open `examples/solr/conf/schema.xml` and edit the "text" field line so that stored is true, as shown in listing 15.

Listing 15: Editing schema.xml to make text fields stored

```
<!-- catchall field, containing all other searchable text fields (implemented
     via copyField further on in this schema  -->
<field name="text" type="text_general" indexed="true" stored="true" multiValued="true"/>
```

After this, shutdown and restart the Solr server and try adding another document via curl as in listing 12.

```
curl "http://localhost:8983/solr/update/extract?literal.id=doc2&commit=true" -F "myfile=@Midterm
```

And it can be queried using the Solr select handler as shown below.

```
wget -O - http://localhost:8983/solr/select?q=relational
```

Term Highlighting in Solr

Now that the text term is being stored, we can also use Solr's hit highlighting feature to return snippets from the text the matched search terms. To have terms be highlighted, add `hl=true&hl.fl=text` to the end of the select URL. See listing 16 for an example. (Note that `fl=id` was added to restrict the returned fields to just the document id to reduce clutter.) The response is shown in listing 17. Notice how context surrounding the first match was returned in the "highlighting" section of the XML. Further details on highlighting options are available at [8].

Listing 16: Hit highlighting with Solr

```
wget -O - "http://localhost:8983/solr/select?q=lucene&fl=id&hl=true&hl.fl=text"
```

Listing 17: XML response to listing 16

```
<?xml version="1.0" encoding="UTF-8"?>
<response>
<lst name="responseHeader">
<int name="status">0</int>
<int name="QTime">29</int>
<lst name="params">
<str name="fl">id</str>
<str name="hl.fl">text</str>
<str name="hl">true</str>
<str name="q">lucene</str>
</lst></lst>
<result name="response" numFound="2" start="0">
<doc>
    <str name="id">SOLR1000</str>
</doc>
<doc>
    <str name="id">doc1</str>
</doc>
</result>
<lst name="highlighting">
<lst name="SOLR1000"/>
<lst name="doc1">
<arr name="text">
    <str>The Apache Foundations &lt;em&gt;Lucene&lt;/em&gt; and Solr Tools
  Ian Will
  May</str>
</arr>
```

```
</lst></lst>
</response>
```

Summary

This paper has described the basic architecture of Apache's powerful Lucene and Solr projects, and has provided working examples for a few common use cases. Lucene's Java API has been discussed, along with two example–one using spatial filtering and displaying results on a map, another indexing PDF and MS Word documents and displaying highlighted results. It has also provided an overview of Solr and has followed Apache's Solr tutorial with deviations to discuss indexing and searching PDF and MS Word documents and hit highlighting. Hands on code and configuration examples have been provided, and supporting runnable examples for the two Lucene demos are available. Hopefully the basics covered here will allow quick incorporation of Lucene or Solr into your next project.

References

[1] *Apache Lucene*. URL: http://wiki.apache.org/solr/HighlightingParameters.

[2] *Apache PDFBox Java Library*. URL: https://pdfbox.apache.org/.

[3] *Apache Solr*. URL: http://lucene.apache.org/solr/.

[4] *Apache Tika*. 2012. URL: http://tika.apache.org/.

[5] P. Carlson. *Apache Lucene - Query Parser Syntax*. 2006. URL: http://lucene.apache.org/core/old_versioned_docs/versions/3_2_0/queryparsersyntax.pdf.

[6] *ExtractingRequestHandler - Solr Wiki*. URL: http://wiki.apache.org/solr/ExtractingRequestHandler.

[7] E. Hatcher. *QueryParser Rules*. Nov. 2003. URL: http://today.java.net/pub/a/today/2003/11/07/QueryParserRules.html.

[8] *HighlightingParameters - Solr Wiki*. URL: http://wiki.apache.org/solr/HighlightingParameters.

[9] G. Ingersoll. *Location-aware search with Apache Lucene and Solr*. Jan. 2012. URL: http://www.ibm.com/developerworks/opensource/library/j-spatial/.

[10] *[PDFBOX-650] Remove dependency on lucene*. URL: https://issues.apache.org/jira/browse/PDFBOX-650.

[11] M. F. Porter. "An algorithm for suffix stripping". In: *Program* 14.3 (1980), pp. 130–137.

[12] J. Reece. *A Concice Tika/Lucene Content Parsing and Indexing Example*. 2011. URL: http://johnreece.com/wordpress/2011/03/13/a-concise-tikalucene-content-parsing-and-indexing-example/.

[13] *Searching Numerical Fields*. Oct. 2009. URL: http://wiki.apache.org/lucene-java/SearchNumericalFields.

[14] Y. Seeley. *Full-Text Search with Lucene*. May 2007. URL: http://people.apache.org/~yonik/presentations/lucene_intro.pdf.

[15] *Solr Tutorial*. 2012. URL: http://lucene.apache.org/solr/api/doc-files/tutorial.html.

[16] *U.S. Board on Geographic Names BGN: Domestic Names*. 2012. URL: http://geonames.usgs.gov/domestic/index.html.