Hiring? Toptal handpicks **top database developers** to suit your needs.

- Start hiring
- Login

- Top 3%
- Why
- Clients
- Partners
- Community
- Blog
- About Us
- Start hiring
- Apply as a Developer
- Login
- Questions?
  - Contact Us

Search Topics
Hire a developer

Full Text Search of Dialogues with Apache Lucene: A Tutorial

View all articles

by **Doug Sparling** - Freelance Software Engineer @ Toptal

#ApacheLucene #FullTextSearch #Indexing #Java

- 822shares

Apache Lucene is a Java library used for the full text search of documents, and is at the core of search servers such as Solr and Elasticsearch. It can also be embedded into Java applications, such as Android apps or web backends.

While Lucene's configuration options are extensive, they are intended for use by database developers on a generic corpus of text. If your documents have a specific structure or type of content, you can take advantage of either to improve search quality and query capability.

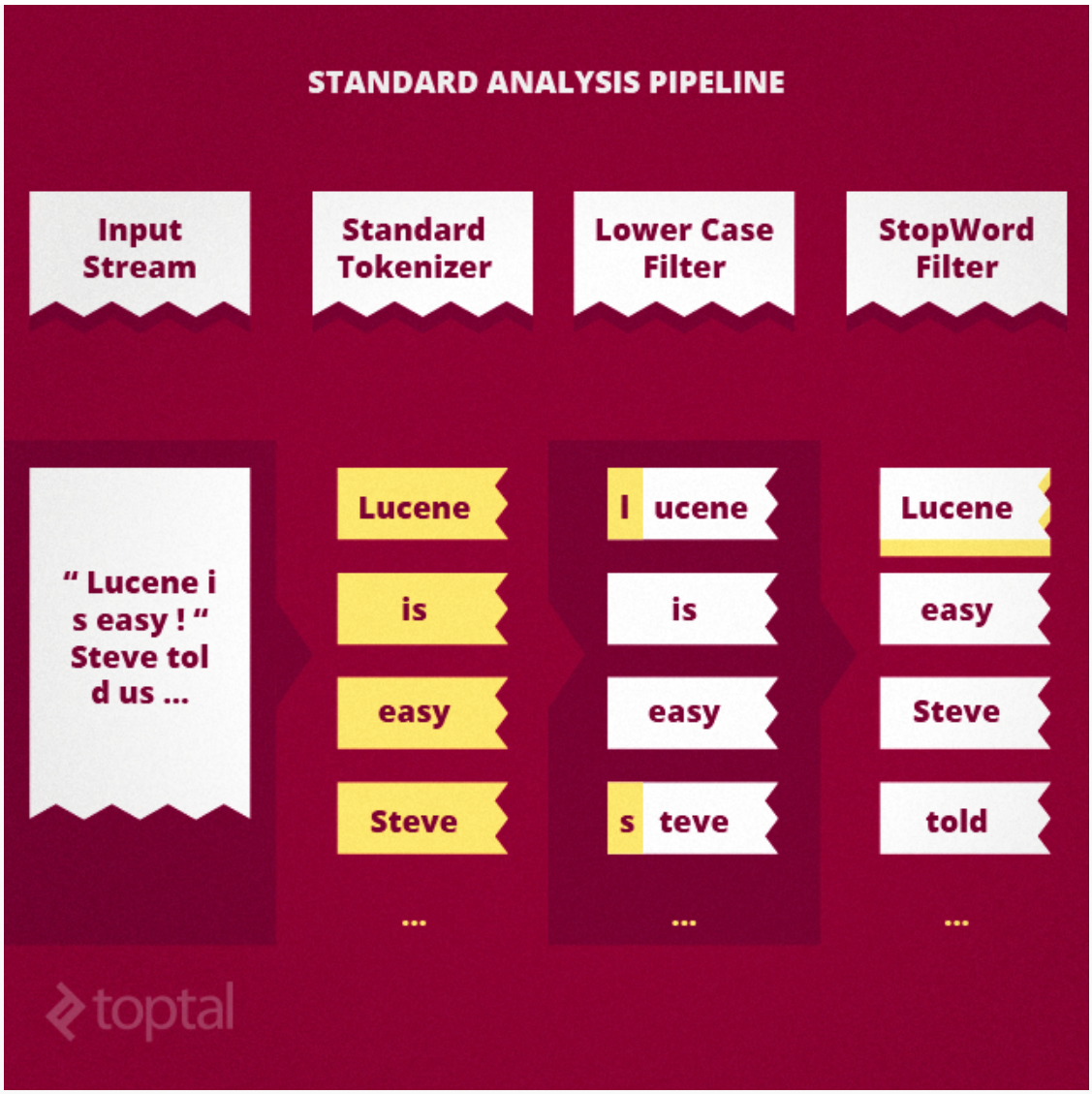
As an example of this sort of customization, in this Lucene tutorial we will index the corpus of Project Gutenberg, which offers thousands of free e-books. We know that many of these books are novels. Suppose we are especially interested in the *dialogue* within these novels. Neither Lucene, Elasticsearch, nor Solr provides out-of-the-box tools to identify content as dialogue. In fact, they will throw away punctuation at the earliest stages of text analysis, which runs counter to being able to identify portions of the text that are dialogue. So it is therefore in these early stages where our customization must begin.

Pieces of the Apache Lucene Analysis Pipeline

The Lucene analysis JavaDoc provides a good overview of all the moving parts in the text analysis pipeline.

At a high level, you can think of the analysis pipeline as consuming a raw stream of characters at the start and producing “terms”, roughly corresponding to words, at the end.

The standard analysis pipeline can be visualized as such:



We will see how to customize this pipeline to recognize regions of text marked by double-quotes, which I will call dialogue, and then bump up matches that occur when searching in those regions.

Reading Characters

When documents are initially added to the index, the characters are read from a Java InputStream, and so they can come from files, databases, web service calls, etc. To create an index for Project Gutenberg, we download the e-books, and create a small application to read these files and write them to the index. Creating a Lucene index and reading files are well travelled paths, so we won't explore them much. The essential code for producing an index is:

```
IndexWriter writer = ...;

BufferedReader reader = new BufferedReader(new InputStreamReader(... fileInputStream ...));

Document document = new Document();
document.add(new StringField("title", fileName, Store.YES));
document.add(new TextField("body", reader));

writer.addDocument(document);
```

We can see that each e-book will correspond to a single Lucene `Document` so, later on, our search results will be a list of matching books. `Store.YES` indicates that we store the *title* field, which is just the filename. We don't want to store the *body* of the ebook, however, as it is not needed when searching and would only waste disk space.

The actual reading of the stream begins with `addDocument`. The `IndexWriter` pulls tokens from the end of the pipeline. This pull proceeds back through the pipe until the first stage, the `Tokenizer`, reads from the `InputStream`.

Also note that we don't close the stream, as Lucene handles this for us.

Tokenizing Characters

The Lucene StandardTokenizer throws away punctuation, and so our customization will begin here, as we need to preserve quotes.

The documentation for `StandardTokenizer` invites you to copy the source code and tailor it to your needs, but this solution would be unnecessarily complex. Instead, we will extend `CharTokenizer`, which allows you to specify characters to "accept", where those that are not "accepted" will be treated as delimiters between tokens and thrown away. Since we are interested in words and the quotations around them, our custom Tokenizer is simply:

```
public class QuotationTokenizer extends CharTokenizer {
        @Override
        protected boolean isTokenChar(int c) {
                return Character.isLetter(c) || c == '"';
        }
}
```

Given an input stream of `[He said, "Good day".]`, the tokens produced would be `[He]`, `[said]`, `["Good]`, `[day"]`

Note how the quotes are interspersed within the tokens. It is possible to write a `Tokenizer` that produces separate tokens for each quote, but `Tokenizer` is also concerned with fiddly, easy-to-screw-up details such as buffering and scanning, so it is best to keep your `Tokenizer` simple and clean up the token stream further along in the pipeline.

Splitting Tokens using Filters

After the tokenizer comes a series of `TokenFilter` objects. Note, incidentally, that *filter* is a bit of a misnomer, as a `TokenFilter` can add, remove, or modify tokens.

Many of the filter classes provided by Lucene expect single words, so it won't do to have our mixed word-and-quote tokens flow into them. Thus, our Lucene tutorial's next customization must be the introduction of a filter that will clean up the output of `QuotationTokenizer`.

This cleanup will involve the production of an extra *start quote* token if the quote appears at the beginning of a word, or an *end quote* token if the quote appears at the end. We will put aside the handling of single quoted words for simplicity.

Creating a `TokenFilter` subclass involves implementing one method: `incrementToken`. This method must call `incrementToken` on the previous filter in the pipe, and then manipulate the results of that call to perform whatever work the filter is responsible for. The results of `incrementToken` are available via `Attribute` objects, which describe the current state of token processing. After our implementation of `incrementToken` returns, it is expected that the attributes have been manipulated to setup the token for the next filter (or the index if we are at the end of the pipe).

The attributes we are interested in at this point in the pipeline are:

- `CharTermAttribute`: Contains a `char[]` buffer holding the characters of the current token. We will need to manipulate this to remove the quote, or to produce a quote token.
- `TypeAttribute`: Contains the "type" of the current token. Because we are adding start and end quotes to the token stream, we will introduce two new types using our filter.
- `OffsetAttribute`: Lucene can optionally store references to the location of terms in the original document. These references are called "offsets", which are just start and end indices into the original character stream. If we change the buffer in `CharTermAttribute` to point to just a substring of the token, we must adjust these offsets accordingly.

You may be wondering why the API for manipulating token streams is so convoluted and, in particular, why we can't just do something like `String#split` on the incoming tokens. This is because Lucene is designed for high-speed, low-overhead indexing, whereby the built-in tokenizers and filters can quickly chew through gigabytes of text while using only megabytes of memory. To achieve this, few or no allocations are done during tokenization and filtering, and so the `Attribute` instances mentioned above are intended to be allocated once and reused. If your tokenizers and filters are written in this way, and minimize their own allocations, you can customize Lucene without compromising performance.

With all that in mind, let's see how to implement a filter that takes a token such as `["Hello]`, and produces the two tokens, `["]` and `[Hello]`:

```
public class QuotationTokenFilter extends TokenFilter {

        private static final char QUOTE = '"';
        public static final String QUOTE_START_TYPE = "start_quote";
        public static final String QUOTE_END_TYPE = "end_quote";

        private final OffsetAttribute offsetAttr = addAttribute(OffsetAttribute.class);
        private final TypeAttribute typeAttr = addAttribute(TypeAttribute.class);
        private final CharTermAttribute termBufferAttr = addAttribute(CharTermAttribute.class);
```

We start by obtaining references to some of the attributes that we saw earlier. We suffix the field names with "Attr" so it will be clear later when we refer to them. It is possible that some `Tokenizer` implementations do not provide these attributes, so we use `addAttribute` to get our references. `addAttribute` will create an attribute instance if it is missing, otherwise grab a shared reference to the attribute of that type. Note that Lucene does not allow multiple instances of the same attribute type at once.

```
private boolean emitExtraToken;
private int extraTokenStartOffset, extraTokenEndOffset;
private String extraTokenType;
```

Because our filter will introduce a new token that was not present in the original stream, we need a place to save the state of that token between calls to `incrementToken`. Because we're splitting an existing token into two, it is enough to know just the offsets and type of the new token. We also have a flag that tells us whether the next call to `incrementToken` will be emitting this extra token. Lucene actually provides a pair of methods, `captureState` and `restoreState`, which will do this for you. But these methods involve the allocation of a `State` object, and can actually be trickier than simply managing that state yourself, so we'll avoid using them.

```
@Override
public void reset() throws IOException {
        emitExtraToken = false;
        extraTokenStartOffset = -1;
        extraTokenEndOffset = -1;
        extraTokenType = null;
        super.reset();
}
```

As part of its aggressive avoidance of allocation, Lucene can reuse filter instances. In this situation, it is expected that a call to `reset` will put the filter back into its initial state. So here, we simply reset our extra token fields.

```
@Override
public boolean incrementToken() throws IOException {

        if (emitExtraToken) {
                advanceToExtraToken();
                emitExtraToken = false;
                return true;
        }

        ...
```

Now we're getting to the interesting bits. When our implementation of `incrementToken` is called, we have an opportunity to *not* call `incrementToken` on the earlier stage of the pipeline. By doing so, we effectively introduce a new token, because we aren't pulling a token from the `Tokenizer`.

Instead, we call `advanceToExtraToken` to setup the attributes for our extra token, set `emitExtraToken` to false to avoid this branch on the next call, and then return `true`, which indicates that another token is available.

```
@Override
public boolean incrementToken() throws IOException {

        ... (emit extra token) ...

        boolean hasNext = input.incrementToken();

        if (hasNext) {
                char[] buffer = termBufferAttr.buffer();

                if (termBuffer.length() > 1) {

                        if (buffer[0] == QUOTE) {
                                splitTermQuoteFirst();
```

```
                        } else if (buffer[termBuffer.length() - 1] == QUOTE) {
                                splitTermWordFirst();
                        }
                } else if (termBuffer.length() == 1) {
                        if (buffer[0] == QUOTE) {
                                typeAttr.setType(QUOTE_END_TYPE);
                        }
                }

        }

        return hasNext;
}
```

The remainder of `incrementToken` will do one of three different things. Recall that `termBufferAttr` is used to inspect the contents of the token coming through the pipe:

1. If we've reached the end of the token stream (i.e. `hasNext` is false), we're done and simply return.
2. If we have a token of more than one character, and one of those characters is a quote, we split the token.
3. If the token is a solitary quote, we assume it is an end quote. To understand why, note that starting quotes always appear to the left of a word (i.e., with no intermediate punctuation), whereas ending quotes can follow punctuation (such as in the sentence, `[He told us to "go back the way we came."]`). In these cases, the ending quote will already be a separate token, and so we need only to set its type.

`splitTermQuoteFirst` and `splitTermWordFirst` will set attributes to make the current token either a word or a quote, and setup the "extra" fields to allow the other half to be consumed later. The two methods are similar, so we'll look at just `splitTermQuoteFirst`:

```
private void splitTermQuoteFirst() {
        int origStart = offsetAttr.startOffset();
        int origEnd = offsetAttr.endOffset();

        offsetAttr.setOffset(origStart, origStart + 1);
        typeAttr.setType(QUOTE_START_TYPE);
        termBufferAttr.setLength(1);

        prepareExtraTerm(origStart + 1, origEnd, TypeAttribute.DEFAULT_TYPE);
}
```

Because we want to split this token with the quote appearing in the stream first, we truncate the buffer by setting the length to one (i.e., one character; namely, the quote). We adjust the offsets accordingly (i.e. pointing to the quote in the original document) and also set the type to be a starting quote.

`prepareExtraTerm` will set the `extra*` fields and set `emitExtraToken` to true. It is called with offsets pointing at the "extra" token (i.e., the word following the quote).

The entirety of `QuotationTokenFilter` is available on Github.

As an aside, while this filter only produces one extra token, this approach can be extended to introduce an arbitrary number of extra tokens. Just replace the `extra*` fields with a collection or, better yet, a fixed-length array if there is a limit on the number of extra tokens that can be produced. See `SynonymFilter` and its `PendingInput` inner class for an example of this.

Consuming Quote Tokens and Marking Dialogue

Now that we've gone to all that effort to add those quotes to the token stream, we can use them to delimit sections of dialogue in the text.

Since our end goal is to adjust search results based on whether terms are part of dialogue or not, we need to attach metadata to those terms. Lucene provides `PayloadAttribute` for this purpose. Payloads are byte arrays that are stored alongside terms in the index, and can be read later during a search. This means that our flag will wastefully occupy an entire byte, so additional payloads could be implemented as bit flags to save space.

Below is a new filter, `DialoguePayloadTokenFilter`, which is added to the very end of the analysis pipeline. It attaches the payload indicating whether or not the token is part of dialogue.

```
public class DialoguePayloadTokenFilter extends TokenFilter {

        private final TypeAttribute typeAttr = getAttribute(TypeAttribute.class);
        private final PayloadAttribute payloadAttr = addAttribute(PayloadAttribute.class);

        private static final BytesRef PAYLOAD_DIALOGUE = new BytesRef(new byte[] { 1 });
        private static final BytesRef PAYLOAD_NOT_DIALOGUE = new BytesRef(new byte[] { 0 });

        private boolean withinDialogue;

        protected DialoguePayloadTokenFilter(TokenStream input) {
                super(input);
        }

        @Override
        public void reset() throws IOException {
                this.withinDialogue = false;
                super.reset();
        }

        @Override
        public boolean incrementToken() throws IOException {
                boolean hasNext = input.incrementToken();

                while(hasNext) {
                        boolean isStartQuote = QuotationTokenFilter
                                        .QUOTE_START_TYPE.equals(typeAttr.type());
                        boolean isEndQuote = QuotationTokenFilter
                                        .QUOTE_END_TYPE.equals(typeAttr.type());

                        if (isStartQuote) {
                                withinDialogue = true;
                                hasNext = input.incrementToken();
                        } else if (isEndQuote) {
                                withinDialogue = false;
                                hasNext = input.incrementToken();
                        } else {
                                break;
                        }
                }

                if (hasNext) {
                        payloadAttr.setPayload(withinDialogue ?
                                        PAYLOAD_DIALOGUE : PAYLOAD_NOT_DIALOGUE);
                }

                return hasNext;
        }
}
```

Since this filter only needs to maintain a single piece of state, `withinDialogue`, it is much simpler. A start quote indicates that we are now within a section of dialogue, while an end quote indicates that the section of dialogue has ended. In either case, the quote token is discarded by making a second call to `incrementToken`, so in effect, *start quote* or *end quote* tokens never flow past this stage in the pipeline.

For example, `DialoguePayloadTokenFilter` will transform the token stream:

```
[the], [program], [printed], ["], [hello], [world], ["]`
```

into this new stream:

```
[the][0], [program][0], [printed][0], [hello][1], [world][1]
```

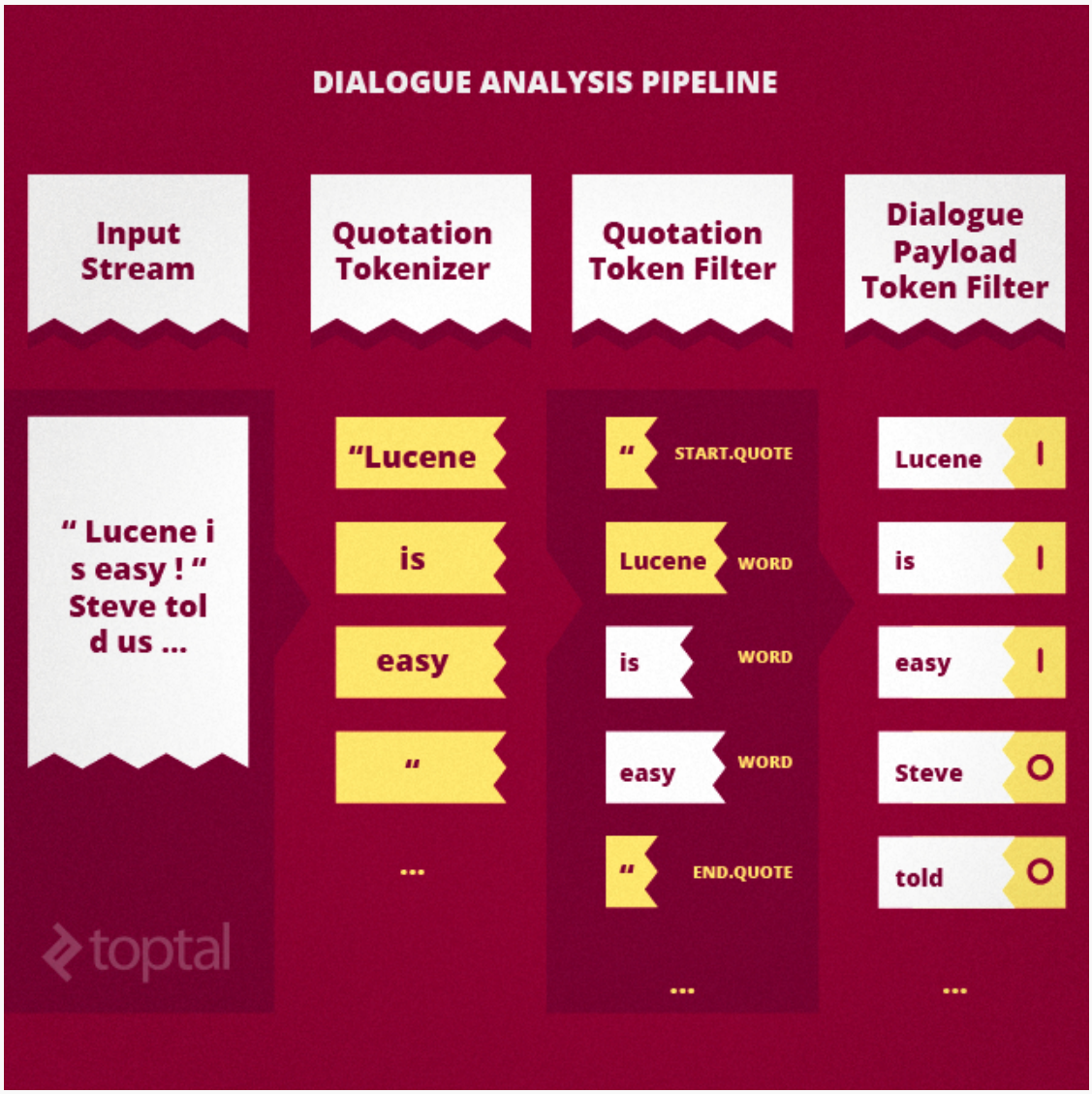
Tying Tokenizers and Filters Together

An `Analyzer` is responsible for assembling the analysis pipeline, typically by combining a `Tokenizer` with a series of `TokenFilters`. `Analyzers` can also define how that pipeline is reused between analyses. We don't need to worry about that as our components don't require anything except a call to `reset()` between uses, which Lucene will always do. We just need to do the assembly by implementing `Analyzer#createComponents(String)`:

```
public class DialogueAnalyzer extends Analyzer {

        @Override
        protected TokenStreamComponents createComponents(String fieldName) {

                QuotationTokenizer tokenizer = new QuotationTokenizer();

                TokenFilter filter = new QuotationTokenFilter(tokenizer);
                filter = new LowerCaseFilter(filter);
                filter = new StopFilter(filter, StopAnalyzer.ENGLISH_STOP_WORDS_SET);
                filter = new DialoguePayloadTokenFilter(filter);

                return new TokenStreamComponents(tokenizer, filter);
        }
}
```

As we saw earlier, filters contain a reference back to the previous stage in the pipeline, so that is how we instantiate them. We also slide in a few filters from `StandardAnalyzer`: `LowerCaseFilter` and `StopFilter`. These two must come after `QuotationTokenFilter` to ensure that any quotes have been separated. We can be more flexible in our placement of `DialoguePayloadTokenFilter`, since anywhere after `QuotationTokenFilter` will do. We put it after `StopFilter` to avoid wasting time injecting the dialogue payload into stop words that will ultimately be removed.

Here is a visualization of our new pipeline in action (minus those parts of the standard pipeline that we have removed or already seen):



`DialogueAnalyzer` can now be used as any other stock `Analyzer` would be, and now we can build the index and move on to search.

Like what you're reading?
Get the latest updates first.
Enter your email address
Get Exclusive Updates
No spam. Just great engineering posts.
Like what you're reading?
Get the latest updates first.
Thank you for subscribing!
Check your inbox to confirm subscription. You'll start receiving posts after you confirm.

- 888shares
-
-
-

Full Text Search of Dialogue

If we wanted to only search dialogue, we could have simply discarded all tokens outside of a quotation and we would have been done. Instead, by leaving all of the original tokens intact, we've given ourselves the flexibility to either perform queries that take dialogue into account, or to treat dialogue like any other part of the text.

The basics of querying a Lucene index are well documented. For our purposes, it is enough to know that queries are composed of `Term` objects stuck together with operators such as `MUST` or `SHOULD`, along with match documents based on those terms. Matching documents are then scored based on a configurable `Similarity` object, and those results can be ordered by score, filtered, or limited. For example, Lucene allows us to do a query for the top ten documents that must contain both of the terms `[hello]` and `[world]`.

Customizing search results based on dialogue can be done by adjusting a document's score based on payload. The first extension point for this will be in `Similarity`, which is responsible for weighing and scoring matching terms.

Similarity and Scoring

Queries will, by default, use `DefaultSimilarity`, which weights terms based on how frequently they occur in a document. It is a good extension point for adjusting weights, so we extend it to also score documents based on payload. The method `DefaultSimilarity#scorePayload` is provided for this purpose:

```
public final class DialogueAwareSimilarity extends DefaultSimilarity {

        @Override
        public float scorePayload(int doc, int start, int end, BytesRef payload) {
                if (payload.bytes[payload.offset] == 0) {
                        return 0.0f;
                }
                return 1.0f;
        }
}
```

`DialogueAwareSimilarity` simply scores non-dialogue payloads as zero. As each `Term` can be matched multiple times, it will potentially have multiple payload scores. The interpretation of these scores up to the `Query` implementation.

Pay close attention to the `BytesRef` containing the payload: we must check the byte at `offset`, since we can't assume that the byte array is the same payload we stored earlier. When reading the index, Lucene isn't going to waste memory allocating a separate byte array just for the call to `scorePayload`, so we get a reference into an existing byte array. When coding against the Lucene API, it pays to keep in mind that performance is the priority, well ahead of developer convenience.

Now that we have our new `Similarity` implementation, it must then be set on the `IndexSearcher` used to execute queries:

```
IndexSearcher searcher = new IndexSearcher(... reader for index ...);
searcher.setSimilarity(new DialogueAwareSimilarity());
```

Queries and Terms

Now that our `IndexSearcher` can score payloads, we also have to construct a query that is payload-aware. `PayloadTermQuery` can be used to match a single `Term` while also checking the payloads of those matches:

```
PayloadTermQuery helloQuery = new PayloadTermQuery(new Term("body", "hello"), new AveragePayloadFunction());
```

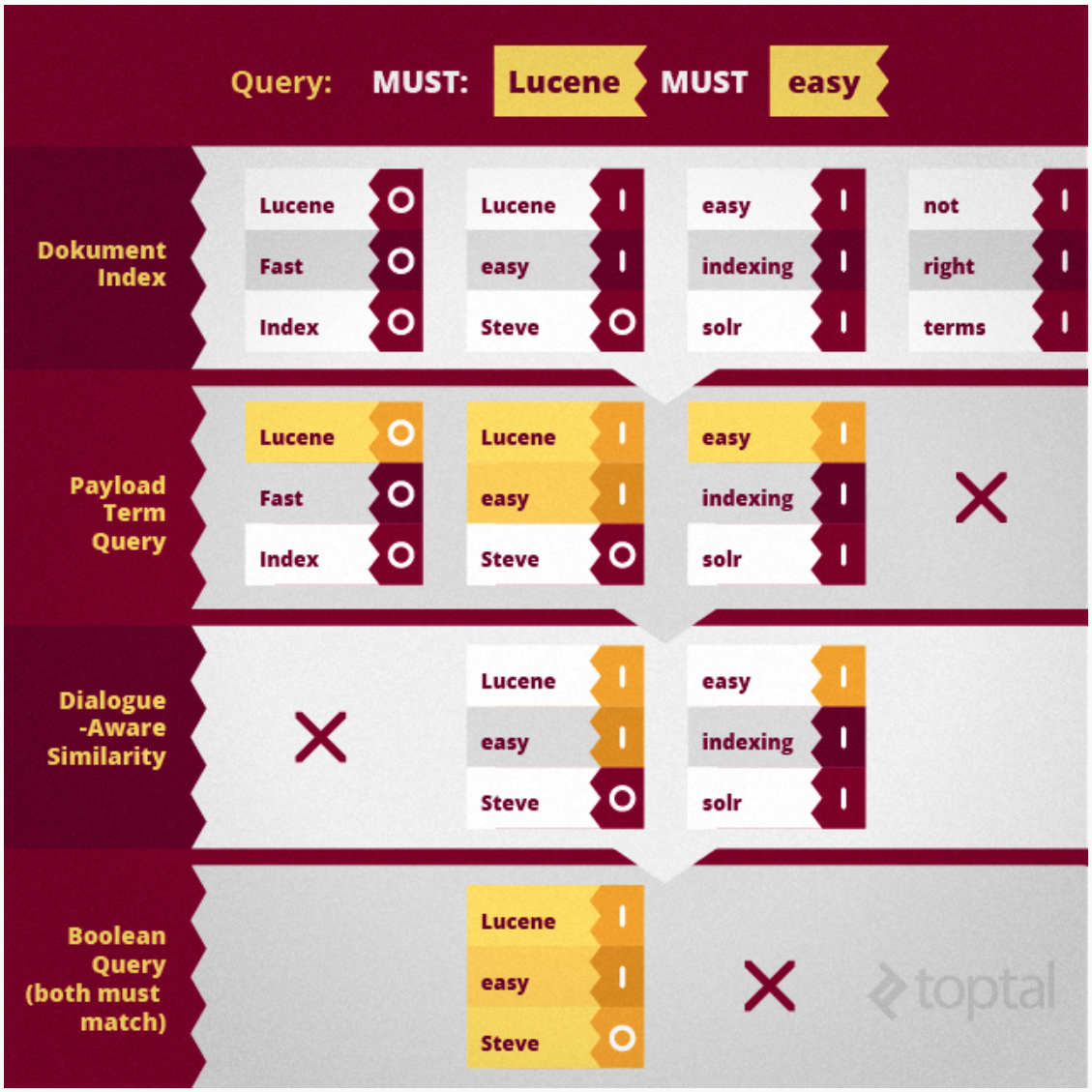
This query matches the term `[hello]` within the *body* field (recall that this is where we put the contents of the document). We must also provide a function to compute the final payload score from all term matches, so we plug in `AveragePayloadFunction`, which averages all payload scores. For example, if the term `[hello]` occurs inside dialogue twice and outside dialogue once, the final payload score will be $\frac{2}{3}$. This final payload score is multiplied with the one provided by `DefaultSimilarity` for the entire document.

We use an average because we would like to de-emphasize search results where many terms appear outside of dialogue, and to produce a score of zero for documents without any terms in dialogue at all.

We can also compose several `PayloadTermQuery` objects using a `BooleanQuery` if we want to search for multiple terms contained in dialogue (note that the order of the terms is irrelevant in this query, though other query types are position-aware):

```
PayloadTermQuery worldQuery = new PayloadTermQuery(new Term("body", "world"), new AveragePayloadFunction());

BooleanQuery query = new BooleanQuery();
query.add(helloQuery, Occur.MUST);
query.add(worldQuery, Occur.MUST);
```

When this query is executed, we can see how the query structure and similarity implementation work together:



Query Execution and Explanation

To execute the query, we hand it off to the `IndexSearcher`:

```
TopScoreDocCollector collector = TopScoreDocCollector.create(10);
searcher.search(query, new PositiveScoresOnlyCollector(collector));
TopDocs topDocs = collector.topDocs();
```

`Collector` objects are used to prepare the collection of matching documents.

collectors can be composed to achieve a combination of sorting, limiting, and filtering. To get, for example, the top ten scoring documents that contain at least one term in dialogue, we combine `TopScoreDocCollector` and `PositiveScoresOnlyCollector`. Taking only positive scores ensures that the zero score matches (i.e., those with no terms in dialogue) are filtered out.

To see this query in action, we can execute it, then use `IndexSearcher#explain` to see how individual documents were scored:

```
for (ScoreDoc result : topDocs.scoreDocs) {
        Document doc = searcher.doc(result.doc, Collections.singleton("title"));

        System.out.println("--- document " + doc.getField("title").stringValue() + " ---");
        System.out.println(this.searcher.explain(query, result.doc));
}
```

Here, we iterate over the document IDs in the `TopDocs` obtained by the search. We also use `IndexSearcher#doc` to retrieve the title field for display. For our query of `"hello"`, this results in:

```
--- Document whelv10.txt ---
0.072256625 = (MATCH) btq, product of:
  0.072256625 = weight(body:hello in 7336) [DialogueAwareSimilarity], result of:
    0.072256625 = fieldWeight in 7336, product of:
      2.345208 = tf(freq=5.5), with freq of:
        5.5 = phraseFreq=5.5
      3.1549776 = idf(docFreq=2873, maxDocs=24796)
      0.009765625 = fieldNorm(doc=7336)
  1.0 = AveragePayloadFunction.docScore()

--- Document daved10.txt ---
0.061311778 = (MATCH) btq, product of:
  0.061311778 = weight(body:hello in 6873) [DialogueAwareSimilarity], result of:
    0.061311778 = fieldWeight in 6873, product of:
      3.3166249 = tf(freq=11.0), with freq of:
        11.0 = phraseFreq=11.0
      3.1549776 = idf(docFreq=2873, maxDocs=24796)
      0.005859375 = fieldNorm(doc=6873)
  1.0 = AveragePayloadFunction.docScore()

...
```

Although the output is laden with jargon, we can see how our custom `Similarity` implementation was used in scoring, and how the `MaxPayloadFunction` produced a multiplier of `1.0` for these matches. This implies that the payload was loaded and scored, and all matches of `"Hello"` occurred in dialogue, and so these results are right at the top where we expect them.

It is also worth pointing out that the index for Project Gutenberg, with payloads, comes to nearly four gigabytes in size, and yet on my modest development machine, queries occur instantaneously. We have not sacrificed any speed to achieve our search goals.

Wrapping Up

Lucene is a powerful, built-for-purpose full text search library that takes a raw stream of characters, bundles them into tokens, and persists them as terms in an index. It can quickly query that index and provide ranked results, and provides ample opportunity for extension while maintaining efficiency.

By using Lucene directly in our applications, or as part of a server, we can perform full text searches in real-time over gigabytes of content. Moreover, by way of custom analysis and scoring, we can take advantage of domain-specific features in our documents to improve the relevance of results or custom queries.

Full code listings for this Lucene tutorial are available on GitHub. The repo contains two applications: `LuceneIndexerApp` for building the index, and `LuceneQueryApp` for performing queries.

The corpus of Project Gutenberg, which can be obtained as a disk image via BitTorrent, contains plenty of books worth reading (either with Lucene, or just the old fashioned way).

Happy indexing!

About the author