Lucene

- Lucene spans
- Lucene term documents and term positions
- Pure negation query in lucene
- Why are Lucene's stored fields so slow to access
- Lucene
- Compiling Lucene with GCJ

Lucene spans

Introduction

In Lucene, a span is a triple (i.e. 3-tuple) of <document number, start position, end position>. Document numbers start from zero. The positions are term positions, not character positions, and start from zero (i.e. the first token of a field has position 0, the second token has position 1, etc.). Start position is the position of the first term in the span. End position is the position of the last term in the span + 1.

For example, say document 1 has the following terms:

Term This is an Apache Lucene test Position 0 1 2 3 4 5
Then “Apache Lucene” matches a span <1 (document number), 3 (first term’s position), 5 (last term’s position + 1)>.

Using spans

Span queries

You don’t have to deal with spans directly. If you create a span query, Lucene takes care of matching the spans for you. Span query cannot be parsed by the default query parser that ships with Lucene. Instead, you create an instance of SpanQuery and give it to other classes like IndexSearcher.

Direct access

You can also using the Spans class. A Span class represents a series of spans instead of a span. It is also an iterator of spans. The easiest way of creating a Span object is to use the SpanQuery.getSpans method, which returns a series of spans matched by a span query (in the form of a Spans object).

Application

Proximity search

Queries like “phrase A” within a distance of “phrase B” can be done using span queries. The API reference has an example of how to query for something like “‘John Kerry’ near ‘George Bush’”.

Word and phrase counts

The Spans interface makes it very easy to get the document count and occurrence count of a word or a phrase (actually, anything that a SpanQuery can represent) in a document set. It can be done by iterating through a Spans object. You can even count how many times phrase A is within a certain distance of phrase B in the document set.

Performance

A typical span query seems to take about twice as long as a typical phrase query, which in turn takes about twice as long as a term query. If you can get away with using a phrase query or even a term query, you might want to do so. For example, “term A near term B” can be done using a phrase query with a non-zero slop.

Lucene term documents and term positions

Introduction

Term documents

For each term T, there are (doc frequency of the term) tuples of <doc ID, freq of T in this doc>.

This information is stored in the .frq file and accessible via the TermDocs interface.

Term positions

For each term T, there are (doc frequency of the term) tuples of <doc ID, freq of T in this doc, (term freq of T in this doc) counts of positions of T in this doc>.

This information is stored in the .prx file and accessible via the TermPositions interface.

Using the information

Via Query

If you use Query classes, they get and make use of term documents and term position so you do not have to worry about them. Non-span queries other than phrase queries use term documents only. Phrase queries uses term documents + term positions to make sure that a document actually

have the terms, say, right next to each other and in order. This makes phrase queries slower than term queries, i.e. searching for the phrase “southern california” show be slower than searching for required words “southern california”.

Lower-level than queries is the spans API. The Spans class is still higher-level than using TermPositions directly.

Via the interfaces

Note that the document number (returned by doc()) and frequency (returned by freq()) of a TermDocs object is undefined until next() is called the first time. This is unclear from the API reference but I have found this out by experiment.

Pure negation query in lucene

In many information-retrieval system, you can use queries like “term1 AND (NOT term2)” but you cannot use queries like “NOT term2” on their own (e.g. to get only documents that do not contain term2). At least the system returns no result even if some documents do not contain term2. This is because:

- Most of these system uses inverted indices, which are essentially “term => document list” mappings. There is no way to reconstruct a list of documents for “NOT term2” from such a mapping.
- The system can generate a set of all document and then subtract those that contains term2 from the set. This can be resource-intensive (e.g. needs O(doc count) storage and processing time) and can be misused by users.

Despite the difficulties, users of some system might need this kind of query. In Lucene, there are two ways to support it:

- When indexing, create a field (e.g. “exists”) with a constant value across documents (e.g. “true”). Then rewrite the query “NOT term2” into “+exists:true -term2”.
- Combine MatchAllDocsQuery, which matches all document, with BooleanQuery. The code will look like this:

```
// Create a query that matches something like “everything AND NOT term2”
MatchAllDocsQuery everyDocClause = new MatchAllDocsQuery();
TermQuery termClause = new TermQuery(new Term(“text”, “term2”));
BooleanQuery query = new BooleanQuery();
query.add(everyDocClause, BooleanClause.Occur.MUST);
query.add(termClause, BooleanClause.Occur.MUST_NOT);
…
IndexSearcher searcher = new IndexSearcher(“/path/to/index”);
Hits hits = searcher.search(query);
```

```
// hits contains only documents that do not contain term2
```

Why are Lucene's stored fields so slow to access

Problem

I have a Lucene index that has some large fields (about 50 KB each) and some small fields (about 50 bytes each). I need to access (iterate) one of the small fields for say 1/10 of the documents. For some reason, such operation is very slow, unreasonably so for such a small field.

Cause

Lucene provides a number of “policies” of how to access fields of a document. (See class org.apache.lucene.document.FieldSelector.) They specify when and how fields are loaded from the index. It turns out that the default is to load all fields in the document as soon as a Document is requested by, say IndexReader. (See class org.apache.lucene.index.FieldsReader, in particular, how it implements the doc(n, FieldSelector) function.) Therefore, when you load a small field, the large fields are also loaded, causing performance problem if you repeat the operation many times.

Solution

The class org.apache.lucene.document.FieldSelectorResult provides several “policies” that you can use. The most interesting one w.r.t. our problem is FieldSelectorResult.LAZY_LOAD. It basically specifies that a field is lazily loaded (i.e. loaded only when needed).

To use this policy, create a FieldSelector object.

```
FieldSelector lazyFieldSelector = new FieldSelector() {□public FieldSelectorResult accept(String
```

When you request the document from an IndexReader, pass this object too.

```
IndexReader reader;...// Open the index reader...Document doc = reader.document(docId, lazyFieldc
```

Note that to get the field, use the Document's getFieldable(String) method instead of getField(String). This is according to the API reference.

```
Fieldable fieldable = document.getFieldable(fieldName);String value = fieldable.stringValue();//
```

Solution

Within a document, stored fields are read sequentially. (See Index File Formats.) In theory, accessing the first fields should be faster than reading the last ones.

Fields are ordered and their orders are stored implicity in the .fnm file. The order that the fields are read from should be the same as the order that you create the fields. To gain performance, create frequently used (and small) stored fields first.

Cause

For some reason, this is still slower than indexing the field and then iterate through all the terms in the field. I looked closer and found another bottleneck.

A Lucene index stores the lengths of the fields in terms of character count, not byte count; Also, a character can be more than a byte long. As we have seen, Lucene stores and processes the fields sequentially. Even if it does not load a field, it must read the whole content of a field to get to the next field. If a large field is not loaded but is before a small field that is loaded, the processing time depends on the length of both fields, not just the small ones.

Solution

The problem will not happen if the field you need to iterate is placed before the large fields, and if you ask the FieldSelector to stop at the field you want.

Say you want to iterate only field “field1”. Then create a FieldSelector that only loads field1 and stops at this field. When creating the index, remember to put the large fields after field1.

```
String fieldToIterate = "field1";...FieldSelector lazyFieldSelector = new FieldSelector() {publ
```

The rest of the code should be the same.

Lucene

Lucene – Apache Lucene is a high-performance, full-featured text search engine library written entirely in Java. http://lucene.apache.org/java/docs/

Compiling Lucene with GCJ

Background

Lucene is a open-source search library written in Java.

GCJ is a Java to native-executable compilter. As shown in a LinuxJournal article, using gcj is similar to using gcc.

Versions Used

As of 1/26/08, the latest version of Lucene is 2.3.0. It needs Java SE 1.4 to compile and run. GCJ 3.3.6 doesn't work because apparently, it doesn't support classes in Java 1.4 (e.g. the new regular expressions classes and functions and NIO stuff). GCJ 3.4.5 and 3.4.6 works. I haven't tried version 4.x yet.

Installing Lucene

Get Lucene's binary. You need lucene-core-2.3.0.jar. There is another JAR file lucene-demos-2.3.0.jar that contains the demos, which are useful to check that your copy of GCJ is working.

Installing GCC

You get GCJ by installing GCC. A how-to of installing GCC 3.3.6 can guide you through that. Basically there are four steps in the process (after downloading and uncompressing the source files):

- configure (I used ./configure —prefix= —with-as=/usr/ccs/bin/ —with-ld= —disable-nls

—enable-libgcj

- make bootstrap
- make check (I think this one is optional)
- make install

If you use Windows, you don’t need the steps above. Just get MinGW and install GCC from it.

Testing

Lucene comes with a pair of indexer and searcher in their demo collection. They have a page that explains how to compile and run the demo.

You can compile the indexing with the following command:

```
/path_of_gcj/bin/gcj lucene-core-2.3.0.jar lucene-demos-2.3.0.jar -o indexer --main=org.apache.l
```

That should create an executable file called indexer in the same directory. Run it and it should show the usage and exit.

When you run the indexer, it might complain that “ld.so.1: indexfiles: fatal: libgcj.so.5: open failed: No such file or directory”. Basically, it tries to find this run-time shared library but can’t find it. You’ll have to tell it where to look at. E.g. in bash, use the following command:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path_of_gcj/lib
```

You can compile and run the searching in a similar way.