

Advanced Lucene

Grant Ingersoll
Ozgur Yilmazel

ApacheCon Europe 2007
May 2, 2007

Overview

- What Yonik said...
- Term Vectors
- Queries In Depth
- Performance
 - Indexing
 - Search
 - Documents

Term Vectors

- Relevance Feedback and “More Like This”
- Domain specialization
- Clustering
- Cosine similarity between two documents
- Highlighter
 - Needs offset info

Lucene Term Vectors (TV)

- In Lucene, a `TermFreqVector` is a representation of all of the terms and term counts in a specific `Field` of a `Document` instance
- As a tuple:

```
termFreq = <term, term countD>  
<fieldName, <..., termFreqi, termFreqi+1, ...>>
```

- As Java:

```
public String getField();  
public String [] getTerms();  
public int [] getTermFrequencies();
```

} Parallel Arrays

Creating Term Vectors

- During indexing, create a `Field` that stores Term Vectors:

```
new Field("title", parser.getTitle(),
        Field.Store.YES,
        Field.Index.TOKENIZED,
        Field.TermVector.YES);
```

- Options are:

```
Field.TermVector.YES
```

```
Field.TermVector.NO
```

```
Field.TermVector.WITH_POSITIONS – Token Position
```

```
Field.TermVector.WITH_OFFSETS – Character offsets
```

```
Field.TermVector.WITH_POSITIONS_OFFSETS
```

Accessing Term Vectors

- Term Vectors are acquired from the IndexReader using:

```
TermFreqVector getTermFreqVector(int docNumber,  
                                  String field)  
TermFreqVector[] getTermFreqVectors(int docNumber)
```

- Can be cast to `TermPositionVector` if the vector was created with offset or position information

- `TermPositionVector` API:

```
int[] getTermPositions(int index);  
TermVectorOffsetInfo [] getOffsets(int index);
```

Relevance Feedback

- Expand the original query using terms from documents
- Manual Feedback:
 - User selects which documents are most relevant and, optionally, non-relevant
 - Get the terms from the term vector for each of the documents and construct a new query
 - Can apply boosts based on term frequencies
- Automatic Feedback
 - Application assumes the top X documents are relevant and the bottom Y are non-relevant and constructs a new query based on the terms in those documents
- See *Modern Information Retrieval* by Baeza-Yates, et. al. for in-depth discussion of feedback

Example

- From Demo, SearchServlet.java
- Code to get the top X terms from a TV

```
protected Collection getTopTerms(TermFreqVector tfv, int
                                numTermsToReturn)
{
    String[] terms = tfv.getTerms();//get the terms
    int [] freqs = tfv.getTermFrequencies();//get the frequencies
    List result = new ArrayList(terms.length);
    for (int i = 0; i < terms.length; i++)
    {
        //create a container for the Term and Frequency information
        result.add(new TermFreq(terms[i], freqs[i]));
    }
    Collections.sort(result, comparator);//sort by frequency
    if (numTermsToReturn < result.size())
    {
        result = result.subList(0, numTermsToReturn);
    }
    return result;
}
```

More Like This

- MoreLikeThis.java
 - In contrib/queries
- Find similar Documents to one document
 - Much simpler/easier than Relevance Feedback
 - Can use Term Vectors or re-analyze Document
 - Terms are ranked by TF/IDF
- Many options for filter terms
 - Min doc frequency
 - Max word length
 - stopwords

Queries

- Query Basics
- Tips and Traps
 - BooleanQuery
 - WildcardQuery
 - ConstantScoreQuery
 - FunctionQuery (Solr)
- Spans
- DisjunctionMaxQuery
- Payloads

Query Basics

- **Reminder, Lucene has many Query types**
 - TermQuery, BooleanQuery, ConstantScoreQuery, MatchAllDocsQuery, MultiPhraseQuery, FuzzyQuery, WildcardQuery, RangeQuery, PrefixQuery, PhraseQuery, Span*Query, DisjunctionMaxQuery, **etc.**
- **QueryParser does not produce all Lucene Query types**
- **Many queries “rewrite” to basic queries like TermQuery and BooleanQuery**

Tips and Traps

- BooleanQuery
 - TooManyClausesException
 - Override with
`BooleanQuery.setMaxClauseCount(int)`
 - Culprits:
 - WildcardQuery
 - Uncontained Relevance Feedback
 - “Documents as queries”

Tips and Traps

- `ConstantScoreQuery`
 - Query wraps a `Filter` and returns the same score (the query boost) for every document in the filter
 - Useful when you just want the results of the `Filter`, not a search
- `FunctionQuery` (via Solr)
 - Computes a score based on the value of a Function
 - e.g. Geographic values like latitude and longitude

Span Queries

- Provide info about where a match took place within a document
- `SpanTermQuery` is the building block for more complicated queries
- **Other `SpanQuery` classes:**
 - `SpanFirstQuery`, `SpanNearQuery`,
`SpanNotQuery`, `SpanOrQuery`,
`SpanRegexQuery`

Spans

- The `Spans` object provides document and position info about the current match
- From `SpanQuery`:

- `Spans getSpans(IndexReader reader)`

- **Interface definitions:**

- `boolean next()` //Move to the next match

- `int doc()` //the doc id of the match

- `int start()` //The match start position

- `int end()` //The match end position

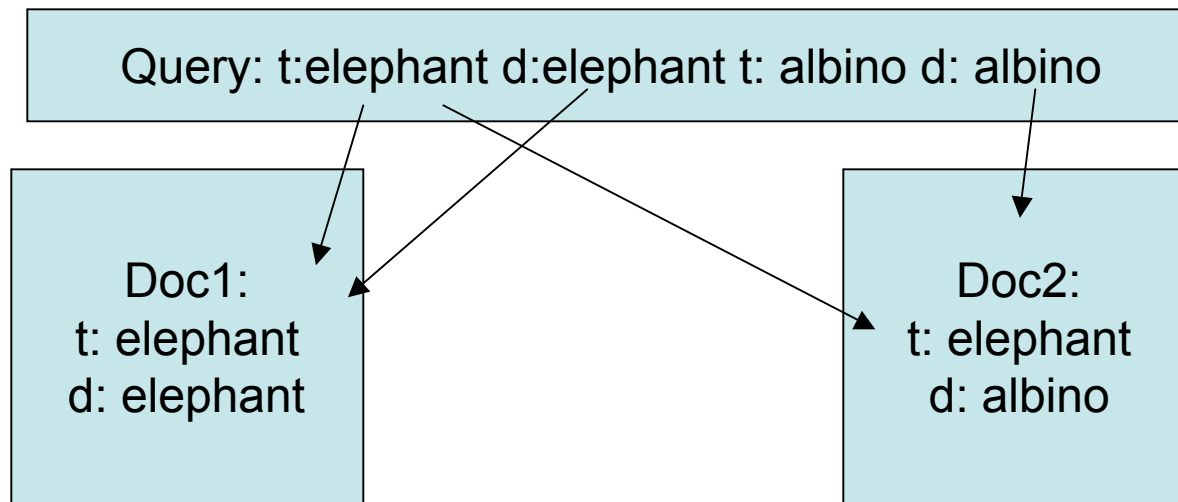
- `boolean skipTo(int target)` //skip to a doc

Phrase Matching using Spans

- `SpanNearQuery` provides functionality similar to `PhraseQuery`
- Use position distance instead of edit distance
- Advantages:
 - Less slop is required to match terms
 - Can be built up of other `SpanQuery` instances

DisjunctionMaxQuery

- Useful when searching across multiple fields
- Example (thanks to Chuck Williams)



- Each Doc scores the same for BooleanQuery
- DisjunctionMaxQuery scores Doc2 higher

Payloads

- Recent addition to Lucene
 - Not Released yet
- Store information at the term level during indexing
 - `Token.setPayload(Payload payload)`
- Will have many `Query` classes that mirror common `Query` classes
 - Work still being done to define capabilities
 - `BoostingTermQuery`
- Override `Similarity` to customize scoring

Payload Uses

- Term level scoring boosts based on Payload
 - Part of Speech
 - Score noun term matches higher than verbs
 - Links/URL
 - Importance, other weights

Performance

- Benchmarker
 - contrib/benchmark in Lucene source
- Consider Lucene 2.1.x
- Indexing
 - What parameters should I set?
 - What about threading?
- Searching
 - Basics of Query Performance
- Documents
 - FieldSelector

contrib/benchmark

- Defines standard corpus, queries and algorithms for benchmarking tasks
- Extensible and Flexible
 - Task based approach allows for new benchmarks
- Provides many services for reporting and accumulating statistics and iterating tasks

Indexing Performance Factors

- Behind the Scenes
 - Lucene indexes `Documents` into memory
 - At certain trigger points, memory (segments) are flushed to the `Directory`
 - Segments are periodically merged
- Much discussion in recent months on optimizing merge approaches
 - Stay Tuned

Indexing Performance Factors

- Analyzer
 - More complicated analysis, slower indexing
- `IndexWriter.setMaxBufferedDocs`
 - Minimum # of docs before merge occurs and a new segment is created
 - Usually, Larger == faster, but more RAM
- `IndexWriter.setMergeFactor`
 - How often segments are merged
 - Smaller == less RAM, better for incremental updates
 - Larger == faster, better for batch indexing
- `IndexWriter.setMaxFieldLength`
 - Limit the number of terms in a Document

Index Threading

- `IndexWriter` is thread-safe
- One open `IndexWriter` per `Directory`
- Parallel Indexing
 - Index to separate `Directory` instances
 - Merge using `IndexWriter.addIndexes`
 - Could also distribute and collect

Other Indexing Factors

- NFS
 - “proceed with caution”
 - Have been recent improvements for Lucene on NFS
- Index Locally and then Replicate
 - See Solr
- Reality check your business needs
 - Many user questions concerning Lucene “performance” on java-user have to do with things outside of Lucene’s control such as XML parsing, etc.
 - If still slow, profile!

Search Performance

- Search speed is based on a number of factors:
 - Query Type(s)
 - Query Size
 - Analysis
 - Occurrences of Query Terms
 - Optimized Index
 - Index Size
 - Index type (`RAMDirectory`, `other`)
 - Usual Suspects
 - CPU, Memory, I/O, Business Needs

Query Types

- Be careful with `WildcardQuery` as it rewrites to a `BooleanQuery` containing all the terms that match the wildcards
- Avoid starting a `WildcardQuery` with a wildcard
- Use `ConstantScoreRangeQuery` instead of `RangeQuery`
- Be careful with range queries and dates
 - User mailing list and Wiki have useful tips for optimizing date handling

Query Size

- Stopword removal
- Search an “all” field instead of many fields with the same terms
- Disambiguation
 - May be useful when doing synonym expansion
 - Difficult to automate and may be slower
 - Some applications may allow the user to disambiguate
- Relevance Feedback/More Like This
 - Use most important words
 - “Important” can be defined in a number of ways

Index Factors

- Size
 - Stopword removal
 - Stemming
 - Lucene has a number of stemmers available
 - Light versus Aggressive
 - May prevent fine-grained matches in some cases
 - Not a linear factor (usually) due to index compression
- Type
 - `RAMDirectory` if index will fit in memory
 - `MMapDirectory` in certain cases may perform better
 - Search user mailing list for information

Usual Suspects

- CPU
 - Profile your application
- Memory
 - Examine your heap size, garbage collection approach
- I/O
 - Cache your `Searcher`
 - Define business logic for refreshing based on indexing needs
 - Warm your `Searcher` before going live -- See Solr
- Business Needs
 - Do you really need to support Wildcards?
 - What about date ranges down to the millisecond?

Document Performance

- Common Use Case:
 - Documents contain several small fields containing metadata about Document
 - One or two large Fields containing content or original file stored as bytes
 - Search occurs, Hits are iterated,
 - Documents are retrieved
 - Small, metadata Field values are displayed on results screen
 - User selects one or two results and views full contents

Field Selector

- Prior to version 2.1, Lucene always loaded **all** `Fields` in a `Document`
- `FieldSelector` API addition allows Lucene to skip large `Fields`
 - Options: Load, Lazy Load, No Load, Load and Break, Load for Merge, Size, Size and Break
- Faster for scenario described
- Makes storage of original content more viable

Resources

- <http://lucene.apache.org>
- Mailing List
 - java-user@lucene.apache.org
- CNLP
 - <http://www.cnlp.org>
 - <http://www.cnlp.org/tech/lucene.asp>
- *Lucene In Action*
 - <http://www.lucenebook.com>