Improved Search with Lucene 4.0
NOVA Apache Lucene/Solr Meetup

Robert Muir, Lucid Imagination
robert.muir@lucidimagination.com, 6/14/2012
Introductions

- What: Examine improvements in 4.0
- Who am I?
  - Lucene Committer & PMC Member
  - Lucid Imagination Employee
- Many big changes coming!
- Examine three general areas:
  - Indexing Improvements
  - Search Improvements
  - Performance Improvements
- Future improvements
But first: Lucene 101

- Search engine library
  - Not an application: use Solr
- Open Source
  - Apache Software Foundation
- Documents: collection of Fields
  - What these are is up to you.
  - Not a parser: use Tika
- Inverted Index
  - Terms map to documents
  - Provide search terms: get ranked results
Lucene 4.0

- Not yet released
  - Nearing alpha stage
  - Alpha → Beta → Final
  - Will release in parallel with Solr 4.0

- Modular API
  - Core, Analysis, Autosuggest, ...

- What changed?
Indexing Improvements

- **Codecs**
  - Flexible index format

- **Index DocValues**
  - Efficient per-document lookup values

- **Miscellaneous Improvements**
  - Offsets in postings
  - Binary terms
  - Additional index statistics
Lucene: indexing basics

- Incremental indexing
  - Segment is a mini-index
  - Periodically merged
  - Update = delete + add

- IndexWriter
  - Add/update/delete docs

- IndexReader
  - Snapshot-in-time view
Codecs: Introduction

- Problem: “one size fits all” index format
- How to integrate future improvements?
  - Example: more efficient index compression
  - But need to support old format, too.
- How to optimize for your data?
  - Without digging into the guts of indexer!
- Idea: format should be pluggable
Codecs: Flex API
Codecs: entire index

- Customize the encoding, data structures, files
- Customize all portions of the index
  - PostingsFormat: terms, docs, positions, …
  - LiveDocsFormat: deleted documents
  - StoredFieldsFormat: stored fields
  - TermVectorsFormat: term vectors
  - …
Codecs: Example use case

- Accelerate near-realtime reopen
- Speed up delete by term on ID field
  - “Primary key lookup”
- Idea: optimize ID field for this use case
Codec: postings formats

- Lucene40: Lucene 4.0 index format
- Pulsing: inline low frequency terms' postings
- Memory: loads field entirely into RAM
- Appending: supports append-only filesystem
- SimpleText: plaintext \textit{(not for production!!!!!)}
- Lucene3x: supports Lucene 3.x index format
Codecs: Configuration

Lucene: use Codec class

Solr: specify in solrconfig.xml/schema.xml

```xml
<fieldType name="text_general" class="solr.TextField"
  postingsFormat="SimpleText"/>
```
Codecs: Configuration

```
field features
   term 1
      doc 20
      freq 1
      pos 12
   term 1,200
      doc 22
      freq 1
      pos 112
   term 1.35ghz
      doc 26
      freq 1
      pos 114
"solr/data/index/_0_1.pst" 4272L, 53640C
```
Index DocValues: Introduction

- Problem: lookup a per-document value
  - RAM-resident for things like scoring factors (e.g. pagerank)
  - Disk-resident for things like document versioning

- Existing workarounds in Lucene have limitations
  - Scoring limited to one byte norm
  - FieldCache requires uninversion to construct
  - Stored fields are slow, geared towards summary results

Index DocValues: Features

- New feature, recently added to Lucene

- ✔ External scoring factors
- ✔ Memory-resident and disk-based lookup
- ✔ DocValues sort-by-term
- ✔ Internal scoring factors
- ✔ Grouping support
- ✗ FST sort implementation
- ✗ Independently updatable
- ✗ Integration with Solr
// some scoring factor for my application
FloatDocValuesField field = new FloatDocValuesField("foo");
doc.add(field);
Offsets in Postings

- Problem: search result highlighting
  - Typically a performance bottleneck
- Two strategies today: neither is great
  - Re-analyze
  - Term vectors
- Highlighting is really important
  - How users judge if a result is relevant
  - Who wants a search engine w/o highlighting!
- How can we speed this up?
Current strategy #1: re-analyze

- Analyze documents on-the-fly
  - Same way as at indexing time
  - Map terms back to locations
- Can be ok *if*:
  - Analysis is very simple
  - Documents are tiny
- Can be slow:
  - Complex analysis pipelines
  - Documents are large
Current strategy #2: term vectors

- Inverted index for **each** doc
  - Avoids re-analysis
- Can be ok *if*:
  - Analysis is very complex
  - Lots of disk space
- Can be slow:
  - 3 seeks per document (.tvx, .tvd, .tvf)
New strategy: offsets in postings

- Offsets in the positions file
  - Avoids re-analysis
  - More efficient access/storage

- Space-efficient:
  - ~ 60% less space than term vectors (English news text)
  - Best case: one byte per position

- Fast:
  - I/O regions should be in cache from scoring
  - Preliminary benchmarks: > 10x
Binary Terms

- In Lucene 4.0, index terms are byte[]
  - Do not need to be unicode text
- Example: Localized Sort and Range
  - Previous versions of Lucene: special encoder
  - Lucene 4.0: 50% space savings
Localized Sort/Range Example

Lucene: use `CollationAnalyzer` class

Solr: specify in `schema.xml`

```xml
<fieldtype name="sort_de" class="solr.CollationField" language="de"/>
```
Additional Index Statistics

- New statistics in Lucene 4.0:
  - `totalTermFreq(term)`: number of occurrences
  - `sumTotalTermFreq(field)`: number of tokens
  - `sumDocFreq(field)`: number of postings
  - `docCount(field)`: number of docs with value

- Available from Lucene APIs
- Available from function queries
- Supports additional scoring algorithms...
Search Improvements

- Improved Scoring API
  - Additional Scoring algorithms
  - Distributed Scoring API

- Spellchecking improvements

- Query parsing improvements
Scoring: Introduction

- Problem: “baked-in” vector space model
- How to integrate additional algorithms?
  - Example: Language Models
  - Before 4.0: write custom Queries
  - Before 4.0: track certain statistics yourself
- How to customize for your data?
  - Without digging into the guts of postings lists!
- Idea: separate “matching” from “scoring”
Scoring: additional algorithms

- Okapi BM25
- Language Models
- Divergence from Randomness
- Information-based Models
Scoring: Configuration

Lucene: use *Similarity* class

Solr: specify in schema.xml

```xml
<similarity class="solr.LMDirichletSimilarityFactory">
  <float name="mu">1000</float>
</similarity>
```
Scoring: distributed API

- Statistics API on IndexSearcher
- 64-bit collection/term statistics
  - For distributed collections > 2B docs
- Compatible with all scoring algorithms
- Solr implementation as a patch
  - Designed for both approximate and exact interchange
Spellchecking Improvements

- New DirectSpellChecker
  - No additional index needed
- Better Suggestions
  - Levenshtein vs. n-gram
- Exposes more configuration options
  - Tune to your collection
Spellchecking: Configuration

Lucene: use DirectSpellChecker class

Solr: specify in solrconfig.xml

```xml
<!-- a spellchecker built from a field of the main index -->
<lst name="spellchecker">
  <str name="name">default</str>
  <str name="field">name</str>
  <str name="classname">solr.DirectSolrSpellChecker</str>
...
```
Query parsing Improvements

- Regular expression queries
  - /mycompany.(com|org|net)/
- Specify number of edits for fuzzy
  - foobar~2
- Wildcard escaping
  - crazy\?*
- Range query syntax improvements
  - Mix inclusive/exclusive bounds
  - Support open-ended syntax in Lucene
Performance Improvements

- Concurrent Flushing
- BlockTree term dictionary
- Improved RAM Efficiency
- Faster Filtered Search
Concurrent Flushing

http://people.apache.org/~mikemccand/lucenebench
BlockTree term dictionary

In Lucene 3 this thing is < 1 QPS!
Improved RAM Efficiency

- Lucene 4.0 uses much less memory
  - Terms Index, Suggester, Synonyms: finite-state
  - Fieldcache: packed integers / utf-8

- Example with Wikipedia collection:
  “Memory footprint reduction from 389M to 90M after some off-the wall sorting and faceting”
Faster Filtered Search

- Up to 500% faster execution time for filtered search.
- Especially faster for “Hard” queries like Span queries / Sloppy Phrase Queries / High Frequency Boolean Queries
- Lucene decides between different execution strategies:
  - Conjunction of query and filter (via skipTo)
  - Drive only by Query, but with filter handled like deleted documents.
Future Improvements

- Block Index Compression
  - PFOR-delta, Simple8b, ...
- Position iterators from Scorers
  - Simplify/speed up Highlighting
  - Proximity Scoring
- Structured/Section Scoring (e.g. BM25F)
Conclusion

- Lucene 4.0 will have many improvements, architectural, and API changes.
- A few of these were introduced here:
  - Ability to customize the index format
  - Additional scoring algorithms
  - Performance Improvements
- More are currently under development.
- For more information, look at CHANGES.txt in subversion, JIRA, ...
More Information

- Lucene Website
  - [http://lucene.apache.org](http://lucene.apache.org)

- Users List
  - java-user@lucene.apache.org

- Lucene Revolution presentations
  - [http://www.lucidimagination.com/devzone/events/lucene_revolution_conferences](http://www.lucidimagination.com/devzone/events/lucene_revolution_conferences)

- Contact Info
  - robert.muir@lucidimagination.com
  - [http://www.lucidimagination.com/blog/author/robert-muir](http://www.lucidimagination.com/blog/author/robert-muir)
  - @rcmuir