Lucene 4 - Next generation open source search

Simon Willnauer
Apache Lucene Core Committer & PMC Chair
simonw@apache.org / simon.willnauer@searchworkings.org
Who am I?

- Lucene Core Committer
- Project Management Committee Chair (PMC)
- Apache Member
- BerlinBuzzwords Co-Founder
- Addicted to OpenSource
http://www.searchworkings.org

- Community Portal targeting OpenSource Search

Welcome, we can see you are a newbie – let us show you around... SearchWorkings.org is a community of search professionals looking for a resource where they can discover, share and discuss the latest technologies and topics.

Featured Topics

Free Online Training

- Integrating Solr with JEE applications
  So you have downloaded Solr, configured it, indexed your data and are now ready to integrate it with the rest of your enterprise Java application. For most situations, this process will begin with...

Featured Blog Entry

- The ManifoldCF authorization model
  Getting documents out of a repository and into Solr is only half of the problem, because it is a rare repository that does not attempt to restrict access to individual documents based on a user's... View in Context »

APACHE LUCENE EUROCON Barcelona 2011

It’s time for Apache Lucene EUROCON in Barcelona. A conference aimed at the European Apache Lucene / Solr open source search community. Two key contributors from SearchWorkings.org have been asked to participate and will be speakers at the event.
Agenda

• Flexible Indexing
• IndexDocValues
• DocumentsWriterPerThread (DWTP)
• Automaton Queries
• Random & Pending Improvements
Architecture prior to Lucene 4.0

IndexWriter

IndexReader

Directory

FileSystem
Architecture with Flexible Indexing

IndexWriter

Flex API

Codec

IndexReader

Directory

FileSystem
Lucene 4.0 Codec Layer

Inverted Index  IndexDocValues  Stored Fields  Segment Metadata

Codec

PostingsFormat  DocValuesFormat  FieldsFormat  SegmentInfosFormat

TermsConsumer  DocValuesConsumer  FieldsWriter  SegmentInfosWriter
TermsProducer  DocValuesProducer  FieldsReader  SegmentInfosReader

PostingsConsumer  PostingsProducer
Good news / Bad news

• 90% will never get in touch with this level of Lucene
• the remaining 10% might be researchers :)
• However - configuration options might be worth while

Why is this cool again?
For Backwards Compatibility you know?

Available Codecs

- Lucene 5
- Lucene 4
- Lucene 3

Index

- segment
  - id
    - Lucene 3
  - title
    - Lucene 3

Index

- segment
  - id
    - Lucene 5
  - title
    - Lucene 5

Index Reader

<< read >>

Index Writer

<< merge >>
Postings Format Per Field

- **field: uid**
  - usually 1 doc per uid
  - likely no shared terms
  - needs to be super fast in a NoSQLish environment

- **field: spell**
  - large number of tokenized unique terms
  - spelling correction - no posting list traversal
  - large amount of key lookups

- **field: body**
  - tokenized terms
  - maybe used for spelling correction
  - general document retrieval
PostingsFormat Per Field

- **field: uid**
  - Pulsing - PostingsFormat
  - inlines postings into the term dictionary
  - inlining is configurable
  - safes additional lookup on disk

- **field: spell**
  - Memory - PostingsFormat
  - loads terms & postings into RAM
  - linear scanning vs. skipping
  - in-mem FST usually very compact

- **field: body**
  - Default - PostingsFormat
  - very memory efficient
  - terminates early for seekExact
  - uses skipping for postings
Using the right tool for the job.

Switching to Memory Postings Format
Using the right tool for the job.

Speedup with Pulsing Codec

![Graph showing speedup with varying number of random term lookups.](image)
Using the right tool for the job.

Switching to BlockTreeTermIndex
Same extensibility is available for

- Stored Fields
- Segment Infos
- Norms and FieldInfos will be added soon
- IndexDocValues
What is this all about? - Inverted Index

Lucene is basically an inverted index - used to find terms QUICKLY!

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The old night keeper keeps the keep in the town</td>
</tr>
<tr>
<td>2</td>
<td>In the big old house in the big old gown.</td>
</tr>
<tr>
<td>3</td>
<td>The house in the town had the big old keep</td>
</tr>
<tr>
<td>4</td>
<td>Where the old night keeper never did sleep.</td>
</tr>
<tr>
<td>5</td>
<td>The night keeper keeps the keep in the night</td>
</tr>
<tr>
<td>6</td>
<td>And keeps in the dark and sleeps in the light.</td>
</tr>
</tbody>
</table>

Table with 6 documents

<table>
<thead>
<tr>
<th>term</th>
<th>freq</th>
<th>Posting list</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>big</td>
<td>2</td>
<td>2 3</td>
</tr>
<tr>
<td>dark</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>did</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>gown</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>had</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>house</td>
<td>2</td>
<td>2 3</td>
</tr>
<tr>
<td>in</td>
<td>5</td>
<td>&lt;1&gt; &lt;2&gt; &lt;3&gt; &lt;5&gt; &lt;6&gt;</td>
</tr>
<tr>
<td>keep</td>
<td>3</td>
<td>1 3 5</td>
</tr>
<tr>
<td>keeper</td>
<td>3</td>
<td>1 4 5</td>
</tr>
<tr>
<td>keeps</td>
<td>3</td>
<td>1 5 6</td>
</tr>
<tr>
<td>light</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>never</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>night</td>
<td>3</td>
<td>1 4 5</td>
</tr>
<tr>
<td>old</td>
<td>4</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>sleep</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>sleeps</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>the</td>
<td>6</td>
<td>&lt;1&gt; &lt;2&gt; &lt;3&gt; &lt;4&gt; &lt;5&gt; &lt;6&gt;</td>
</tr>
<tr>
<td>town</td>
<td>2</td>
<td>1 3</td>
</tr>
<tr>
<td>where</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>
Intersecting posting lists

Yet, once we found the right terms the game starts....

Posting Lists (document IDs)

<table>
<thead>
<tr>
<th>5</th>
<th>10</th>
<th>11</th>
<th>55</th>
<th>57</th>
<th>59</th>
<th>77</th>
<th>88</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td>13</td>
<td>44</td>
<td>55</td>
<td>79</td>
<td>88</td>
<td>99</td>
</tr>
</tbody>
</table>

AND Query

What goes into the score? PageRank?, ClickFeedback?
How to store scoring factors?

- **Stored Fields**: Yeah - s/ms/s/ in your query response time

- **FieldCache**: Awesome - lets undo all the indexing work!

Problem here: this works well :(
Lucene can un-invert a field into FieldCache

<table>
<thead>
<tr>
<th>weight</th>
<th>term</th>
<th>freq</th>
<th>Posting list</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.8</td>
<td>1.0</td>
<td>1</td>
<td>1 6</td>
</tr>
<tr>
<td>1.0</td>
<td>2.7</td>
<td>1</td>
<td>2 3</td>
</tr>
<tr>
<td>2.7</td>
<td>3.2</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>4.3</td>
<td>4.3</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>7.9</td>
<td>4.7</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>1.0</td>
<td>5.8</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3.2</td>
<td>7.9</td>
<td>1</td>
<td>5 9</td>
</tr>
<tr>
<td>4.7</td>
<td>9.0</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

un-invert
FieldCache - loading

Simple Benchmark

- Indexing **100k**, **1M** and **10M** random floats
- not analyzed no norms
- load field into **FieldCache** from optimized index

<table>
<thead>
<tr>
<th></th>
<th>100k Docs</th>
<th>1M Docs</th>
<th>10M Docs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>122 ms</td>
<td>348 ms</td>
<td>3161 ms</td>
</tr>
</tbody>
</table>

Remember, this is only one field! Some apps have many fields to load to **FieldCache**
The more native solution - IndexDocValues

- A dense column based storage
- 1 value per document
- accepts primitives - no conversion from / to string
  - short, int, long (compressed variants)
  - float & double
  - byte[
- each field has a **DocValues Type** but can still be **indexed** or **stored**
- Entirely **optional**
Simple Layout - even on disk

1 column per field and segment

<table>
<thead>
<tr>
<th>field: time</th>
<th>field: id (searchable)</th>
<th>field: page_rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>1288271631431</td>
<td>1</td>
<td>3.2</td>
</tr>
<tr>
<td>1288271631531</td>
<td>5</td>
<td>4.5</td>
</tr>
<tr>
<td>1288271631631</td>
<td>3</td>
<td>2.3</td>
</tr>
<tr>
<td>1288271631732</td>
<td>4</td>
<td>4.44</td>
</tr>
<tr>
<td>1288271631832</td>
<td>6</td>
<td>6.7</td>
</tr>
<tr>
<td>1288271631932</td>
<td>9</td>
<td>7.8</td>
</tr>
<tr>
<td>1288271632032</td>
<td>8</td>
<td>9.9</td>
</tr>
<tr>
<td>1288271632132</td>
<td>7</td>
<td>10.1</td>
</tr>
<tr>
<td>1288271632233</td>
<td>12</td>
<td>11.0</td>
</tr>
<tr>
<td>1288271632333</td>
<td>14</td>
<td>33.1</td>
</tr>
<tr>
<td>1288271632433</td>
<td>22</td>
<td>0.2</td>
</tr>
<tr>
<td>1288271632533</td>
<td>32</td>
<td>1.4</td>
</tr>
<tr>
<td>1288271632637</td>
<td>100</td>
<td>55.6</td>
</tr>
<tr>
<td>1288271632737</td>
<td>33</td>
<td>2.2</td>
</tr>
<tr>
<td>1288271632838</td>
<td>34</td>
<td>7.5</td>
</tr>
<tr>
<td>1288271632938</td>
<td>35</td>
<td>3.2</td>
</tr>
<tr>
<td>1288271633038</td>
<td>36</td>
<td>3.4</td>
</tr>
<tr>
<td>1288271633138</td>
<td>37</td>
<td>5.6</td>
</tr>
<tr>
<td>1288271633233</td>
<td>38</td>
<td>45.0</td>
</tr>
</tbody>
</table>

integer  integer  float 32
Arbitrary Values - The byte[] variants

- Length Variants:
  - Fixed / Variable

- Store Variants:
  - Straight or Referenced
IndexReader reader;
IndexDocValues docValues = reader.docValues("page_rank");
Source source = docValues.getSource();

loads in RAM on first access

IndexReader reader;
IndexDocValues docValues = reader.docValues("page_rank");
Source source = docValues.getDirectSource();

goes to disk directly
performance hit 40 - 80% (YMMV)
Indexing Ingest Rate over time with Lucene 3.x Indexing 7 Million 4kb wikipedia documents

Question: WTF is the IndexWriter doing there?
A whole lot of nothing.... prior to DWPT

Answer: it gives you threads a break and it’s having a drink with your slow-as-s**t IO System
Keep your resources busy with DWPT

IndexWriter

DocumentsWriter

DWPT DWPT DWPT DWPT DWPT

Flush to Disk

Directory

Multi-Threaded
Indexing Ingest Rate over time with Lucene 4.0 & DWPT Indexing 7 Million 4kb wikipedia documents

vs. 620 sec on 3.x
280% improvement

adjusted some settings (less RAM more Concurrency)

committed DWPT

This might safe you some machines if you have to index a lot of text! I’d be interested in how much we can improve the CO2 footprint with better resource utilization.
Search as a DFA - Automaton Queries

RegExp: (ftp|http).*

Fuzzy: dogs~1

Fuzzy-Prefix: (dogs~1).*
Automaton Queries (Fuzzy)

Example DFA for “dogs” Levenshtein Distance 1

Accepts: “dugs”
Here are the 20k % everybody waits for :D

In Lucene 3 this is about 0.1 - 0.2 QPS
// a term representative of the query, containing the field.
// term text is not important and only used for toString() and such
Term term = new Term("body", "dogs~1");

// builds a DFA for all strings within an edit distance of 2 from "bla"
Automaton fuzzy = new LevenshteinAutomata("dogs").toAutomaton(1);

// concatenate this with another DFA equivalent to the "*" operator
Automaton fuzzyPrefix = BasicOperations.concatenate(fuzzy, BasicAutomata.makeAnyString());

// build a query, search with it to get results.
AutomatonQuery query = new AutomatonQuery(term, fuzzyPrefix);
Random Improvements

• Opaque terms use UTF-8 instead of UTF-16 (Java Strings)
• Memory footprint reduction up to 80% (new DataStructures etc.)
• DeepPaging support
• Direct Spellchecking (using FuzzyAutomaton)
• Additional Scoring models
  • BM25, Language Models, Divergence from Randomness
  • Information Based Models
Pending Improvements

- Block Index Compression (PFOR-delta, Simple*, GroupVInt)
- PositionIterators for Scorers
  - Offsets in PostingLists (fast highlighting)
  - Flexible Proximity Scoring
- Updateable IndexDocValues
- Cut over Norms to IndexDocValues
Thank you for your attention!
Maintaining Superior Quality in Lucene

• Maintaining a Software Library used by thousands of users comes with responsibilities

• Lucene has to provide:
  • Stable APIs
  • Backwards Compatibility

• Needs to prevent performance regression

• Lets see what Lucene does about this.
Tests getting complex in Lucene

- Lucene needs to test
  - 10 different Directory Implementations
  - 8 different Codec Implementation
  - tons of different settings on IndexWriter
  - Unicode Support throughout the entire library
  - 5 different MergePolicies
  - Concurrency & IO
Solution: Randomized Testing

• Each test is initialized with a random seed

• Most tests run with:
  • A random Directory, MergePolicy, IndexWriterConfig & Codec

• # iterations and limits are selected at random

• Open file handles are tracked and test fails if they are not closed

• Tests use Random Unicode Strings (we broke several JVM already)

• On failure, test prints a random seed to reproduce the test
Randomized Testing - the Problem

• You still need to write the test :)  
• Your test can fail at any time  
  • Well better than not failing at all!  
• Failures in concurrent tests are still hard to reproduce even with the same seed
Investing in Randomized testing

• Lucene gained the ability to rewrite large parts of its internal implementations without much fear!

• Found 10 year old bugs in everyday code

• Prevents leaking file handles (random exception testing)

• Gained confidence that if there is a bug we gonna hit it one day