Performance optimization with Lucene 4
Who am I?

- Lucene Core Committer & PMC Member
- Co-Founder ElasticSearch Inc.
- Co-Founder BerlinBuzzwords
- Twitter: @s1m0nw
- simonw@apache.org
- simon.willnauer@elasticsearch.com
Why are you here?

- You are Lucene Expert and curious what you can do tomorrow? - **Check**!
- You are curious how Lucene can even better that what we already have? - **Check**!
- You are an IR - Researcher and need more ways to do crazy shit? - **Check**!
- Every CPU cycle counts, ah one of those? - **Check**!
- You are curious how to gain a better user experience? - **Check**!
What is performance?

- Better search quality? - Precision / Recall etc.?
- Faster query times?
- Less RAM usage?
- Less Disk usage?
- Higher concurrency?
- Less Garbage to collect?
- An excuse to justify to work on cool things? ;)

Tuesday, November 6, 2012
• As usual, it depends!
  • Figure out what are your bottlenecks!
  • Benchmark and make your results repeatable!
  • 10x faster than crazy fast is still crazy fast!

• If you are in doubt:
  • Reduce the variables in you benchmark!
  • You can still tune just for the sake of it!
Lucene 4.0

Flexibility, Speed & Efficiency
• Pluggable Codecs
• Per Document Values (DocValues)
• Concurrent Flushing
• Multiple Scoring Models - flexible ranking
• New Term Dictionary
• From UTF-16 to UTF-8
  • no string objects anymore!
Concurrent Flushing
aka. DocumentsWriterPerThread (DWPT)
Writing Documents in Lucene 3.x

- `IndexWriter`
- `DocumentsWriter`
- `Thread State`
- `Flush to Disk`
- `Directory`
- `merge segments in memory`
- `Merge on flush`
- `Single-Threaded`
- `Multi-Threaded`
A benchmark (10M English Wikipedia)

Trunk No. Threads: 10 RAM Buffer: 1024.0 MB
Directory: NIOFSDirectory numDocs: 10000000
  indexing: 620 sec
  merges: 174 sec.
  commit: 24 sec.
Concurrent Flushing in 4.0

Flush to Disk

Directory

Multi-Threaded

Wednesday, November 6, 2012
The same Benchmark...

DocumentsWriterPerThread No. Threads: 10 RAM Buffer: 1024.0 MB
Directory: NIOFSDirectory numDocs: 10000000
indexing: 260 sec
merges: 92 sec.
commit: 23 sec.
The improvement...

Committed Concurrent Flushing

Reduced RAM buffer (ramBufferSizeMB) from 512MB to 320MB
Increased the # of threads from 6 to 20

http://people.apache.org/~mikemccand/lucenebench/indexing.html
Concurrent Flushing

- Indexing can gain a lot if hardware is concurrent
  - wait free flushing and indexing
- less RAM might increase your throughput
  - maximizing the IO utilization
- Concurrent Flushing can “hammer” your machine
  - if ssh doesn’t respond - it’s DWPT
- More segments are created ie. more merging
- Tune carefully if you index in to search machines
  - you can easily kill you IO cache - 1 indexing thread might be enough!
  - adjust # thread states and the RAM buffer
DocValues

aka. Column Stride Fields
You Know FieldCache?

Lucene can un-invert a field into FieldCache

- Weight
  - 5.8
  - 1.0
  - 2.7
  - 2.7
  - 4.3
  - 7.9
  - 1.0
  - 3.2
  - 4.7
  - 7.9
  - 9.0

- Parse
  - Convert to datatype

- Un-invert
  - Array per field / segment
  - Segment
  - UTF-8 bytes

- Term
  - Freq
  - Posting list
  - 1.0 1 1 6
  - 2.7 1 2 3
  - 3.2 1 7
  - 4.3 1 4
  - 4.7 1 8
  - 5.8 1 0
  - 7.9 1 5 9
  - 9.0 1 10

- Float 32
The problem...

- Uninverting is heavy (CPU & IO)
- Creates potentially lots of garbage
- Required to be in JVM Memory
- NRT suffers on Re-Open
- Warming Queries take forever
- Unnecessary type conversion
- All fields are always sorted!
Once column per field and segment

<table>
<thead>
<tr>
<th>field: time</th>
<th>field: id</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>1288271632333</td>
<td>38</td>
<td>45.0</td>
</tr>
</tbody>
</table>
DocValues

- No Uninverting
- Compact In-Memory representation
- Fast Loading (~10x faster than FC for a float field)
- Strong typed (int, long, float, double, bytes)
- Sorted if necessary
- On-Disc access via same interface
- Possible on any field
- One Value per Document & Field
Usecases

- Sorting
- Grouping
- Faceting
- Scoring (Norms & Document Boosting)
- Key / Value Lookups
- Persisted Filters
- Geo-Search
Flexible Scoring

Similarity & Friends
Lucene 3.x

- Vector-Space Model (TF-IDF) and that’s it
- Hard to extend
- Insufficient index statistics (avg. field length)
- Global model and not per-field
Lucene 4.0

- Added Per-Field Similarity
- Score-calculation is private to the similarity
- Lots of new index statistics
  - total term frequency
  - sum document frequency
  - sum total term frequency
  - doc count per field
- Norms are DocValues ie. not bound to single byte!
New Scoring models

- Okapi BM-25 Model
- Language Models
- Information Based Models
- Divergence from Randomness
- Yours goes here....
Codecs
aka. Pluggable Index Formats
• One index format
• Impossible to extend without forking Lucene
• Improvements hardly possible
  • Backwards Compatibility
  • Tight coupled Reader and Writer
• Even experiments required massive internal Lucene knowledge
Lucene 4.0

• Introduced a Codec Layer
  • a common interface providing access to low-level data-structures
  • all read and write operations & format are private to the codec
  • fully customizable
  • Postings, Term-Dictionary, DocValues, Norms are per field
What does this buy us?

• Data-Structures tailored to a specific usecase
  • Wanna read you document backwards - do it!
  • Wanna keep every term in memory - do it!
  • Wanna use a B-Tree instead of a FST - do it!
  • Wanna use a Bloom Filter on top - do it!

• Lucene gave up control over all low level data-structures

• Lots of different implementations shipped with Lucene 4
Available codecs / formats?

- **Pulsing Postings Format**
  - Inlines postings into the term dictionary

- **Bloom Postings Format**
  - Uses a bloom filter to speed up term lookups
  - Helps with NRT on ID fields to speed up deleting docs

- **Block Postings Format**
  - uses state of the art block compression
  - new default in Lucene 4.1
  - speeds up queries if positions are present but not used
Available codecs / formats?

- **Block Tree Term Index (default Lucene 4.0)**
  - reduces memory footprint 30x less
  - massive lookup speed improvements

- **Simple Text Postings Format**
  - helpful for debugging
  - writes everything as plain text

- **Memory Postings Format**
  - holds everything in memory
  - 1 Million Key-Value lookups / second
Not just postings

• Compressed Stored Fields
  • Will come with Lucene 4.1
  • Uses LZ4 Compression

• Everything we write is exposed via Codec
  • DocValues - have your own format
  • Norms (Essentially DocValues)
  • Delete Documents
  • Term Vectors
  • Segment Level information
Encourage Researchers

- Good idea for postings compression?
  - write a postings format!

- Lucene offers a lot now on the lowest level!
  - you like bits and bytes - help us to improve!

- Try - Measure - Improve!
Wrapping up
What is left?

• ...if I had more time...
• Improved Filter execution up to 500% faster
• Automaton Queries
  • Fast Regular Expression Query
  • FuzzyQuery is 100x to 200x faster than in 3.x
• Term offsets in the index
• New Spellcheckers and Query Suggesters
• Many more... talk to me if you are curious!
Thank You!
• Check out our FST Package
  • Highly memory efficient and Fast Finite State Transducer
  • Excellent for fast key / value lookups
  • Suggesters / TermDictionaries / Analyzers use it

Output of the FST

\[ \text{FST\langle Pair\langle Long, BytesRef\rangle\rangle} \quad \text{fst} \]

Input is a Int 32 sequence (UTF-32) and output a Long / Bytes pair
• Check out the Automaton Package
  • Flexible query creation
  • Combine Levenshtein Automaton other Automatons

```java
// 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);
```