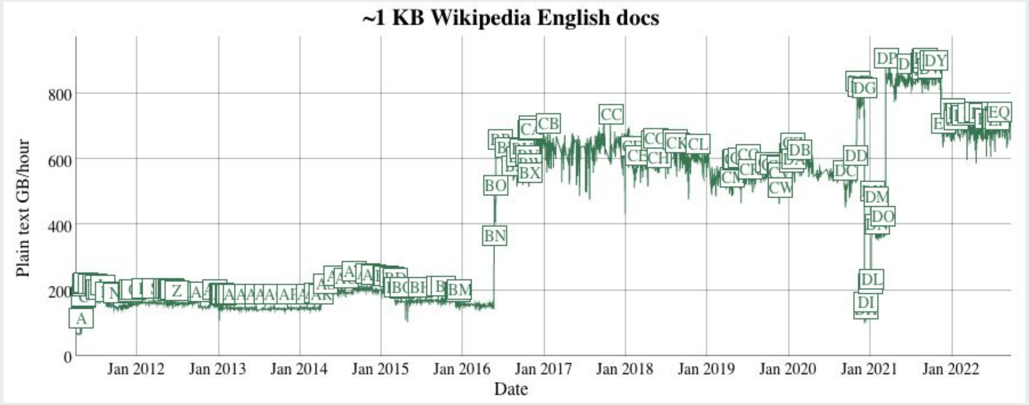
Learning from 11+ years of Apache Lucene™ benchmarks

Mike McCandless
Committer and PMC Member
Apache Lucene

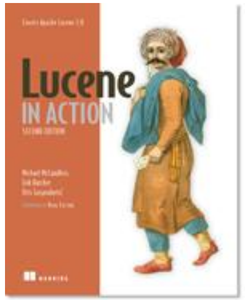


APACHECON NORTH AMERICA
OCTOBER 3 - 6, 2022
NEW ORLEANS
LOUISIANA
WWW.APACHECON.COM

Who am I?

- Lucene committer (16 years) and PMC member, Apache Member
- blog.mikemccandless.com
- Amazon Product Search

@mikemccand at Apache/Twitter/LinkedIn



Outline

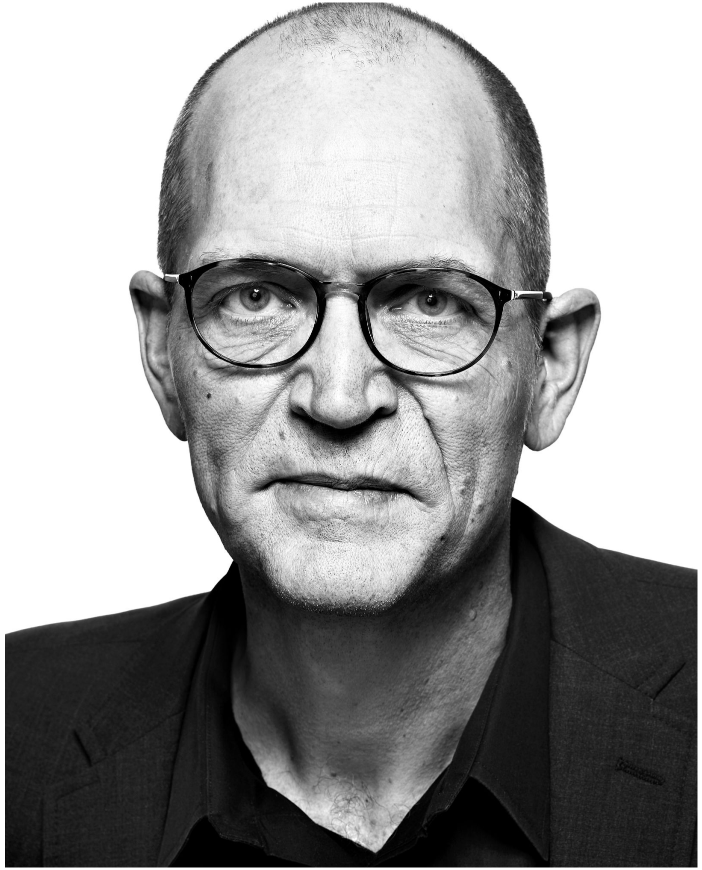
Quick introduction to Apache Lucene

Overview of our benchmark tooling

Battle scars!

Apache Lucene

- High performance Java search engine
- Started in 1999, still active!
- OpenSearch, ElasticSearch, Solr build on Lucene
- **Thank you Doug Cutting!**



reddit

amazon

slack

Walmart

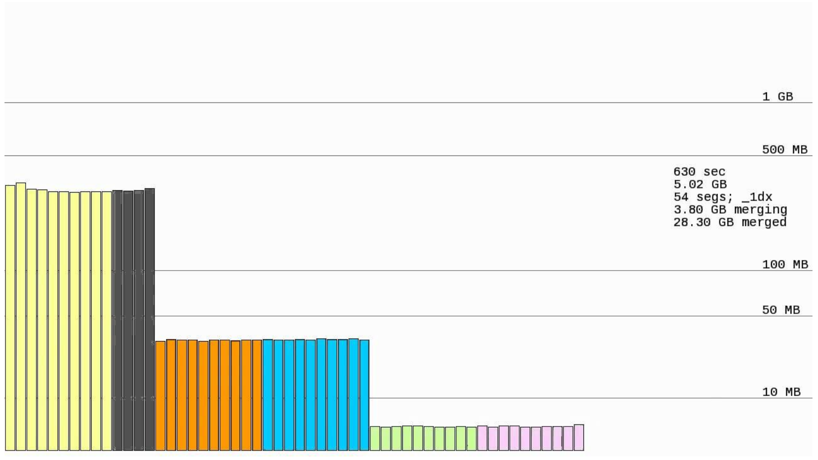
tinder

LinkedIn

wayfair

Indexing and Searching

- Add documents to the index
- Index consists of segments, periodically merged
- Search all segments
- Searching is latency sensitive!
  Typically interactive.
- Indexing (usually) less so
- [Visualizing merges]



Outline

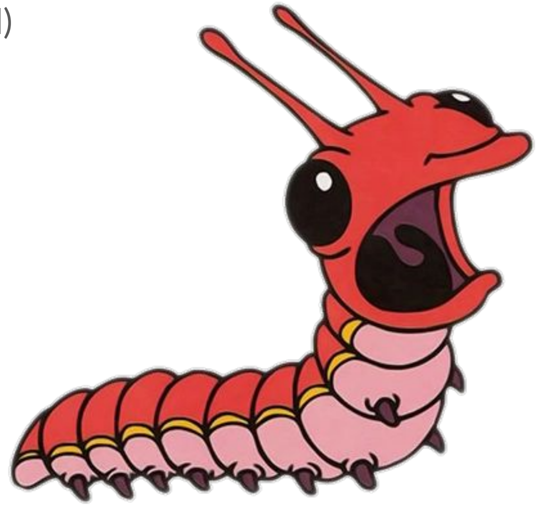
Quick introduction to Apache Lucene

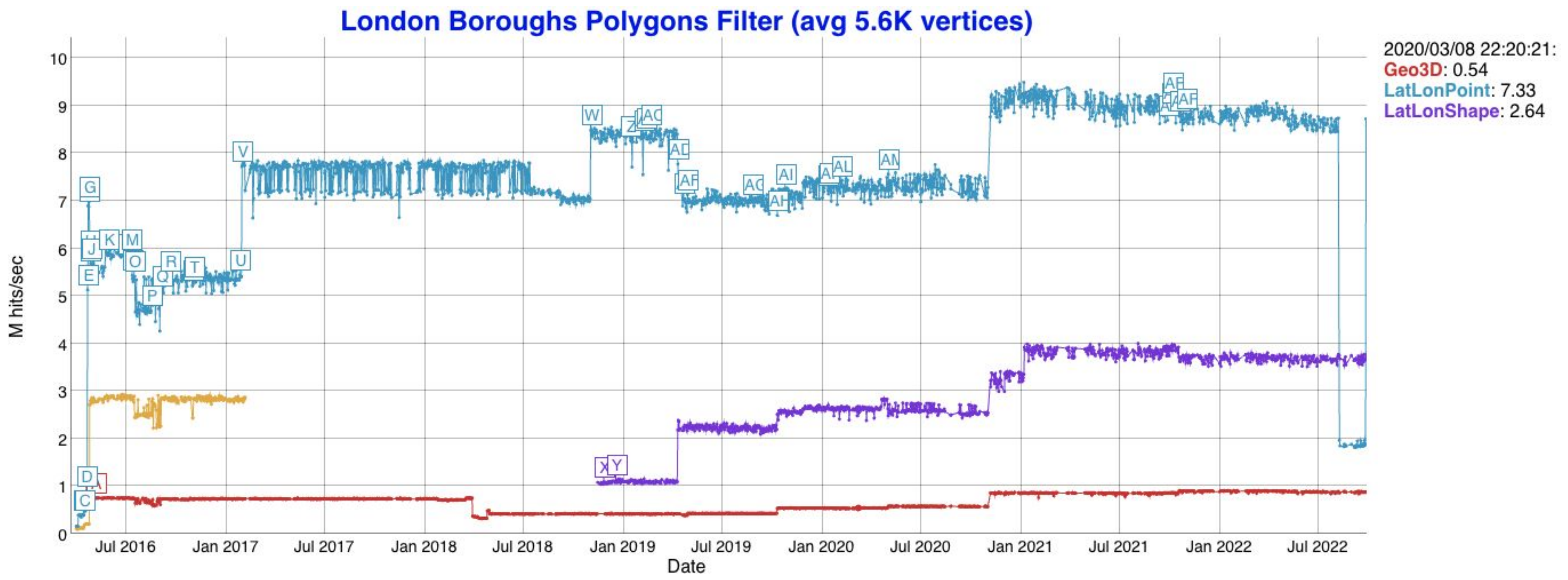
Overview of our benchmark tooling

Battle scars!

Why?

- Catch accidental performance regressions (sudden or gradual)
- Measure performance of a particular code change
- Which compression algo is the best default for stored fields?
- Micro vs macro benchmarks
- Original blog post (2011)





Issue and chart.

Example: testing a code change

```
                         QPS (stddev) base      QPS (stddev) candidate Pct diff               p-value

              OrHighNotMed    674.76       (4.8%)     680.97       (8.0%)    0.9% ( -11% -   14%) 0.659
                  PKLookup    153.45       (4.3%)     155.13       (3.8%)    1.1% (  -6% -    9%) 0.394
                    Fuzzy1     56.57       (9.1%)      57.76       (6.7%)    2.1% ( -12% -   19%) 0.406
     BrowseMonthSSDVFacets     19.59      (10.4%)      20.03       (6.7%)    2.3% ( -13% -   21%) 0.413
  AndHighHighDayTaxoFacets     19.22       (1.6%)      22.13       (2.2%)   15.1% (  11% -   19%) 0.000
   AndHighMedDayTaxoFacets     25.62       (1.5%)      29.93       (2.2%)   16.8% (  12% -   20%) 0.000
      MedTermDayTaxoFacets     12.96       (2.2%)      18.99       (3.4%)   46.5% (  39% -   53%) 0.000
    OrHighMedDayTaxoFacets      3.97       (2.0%)       5.81       (4.3%)   46.5% (  39% -   53%) 0.000
     BrowseMonthTaxoFacets      2.59      (10.9%)      11.16      (35.8%)  330.4% ( 255% -  423%) 0.000
      BrowseDateTaxoFacets      2.44       (9.7%)      13.12      (51.8%)  438.1% ( 343% -  553%) 0.000
 BrowseDayOfYearTaxoFacets      2.44       (9.7%)      13.13      (51.7%)  438.2% ( 343% -  552%) 0.000
```

Explore using SORTED_NUMERIC doc values to encode taxonomy ordinals for faceting

What?

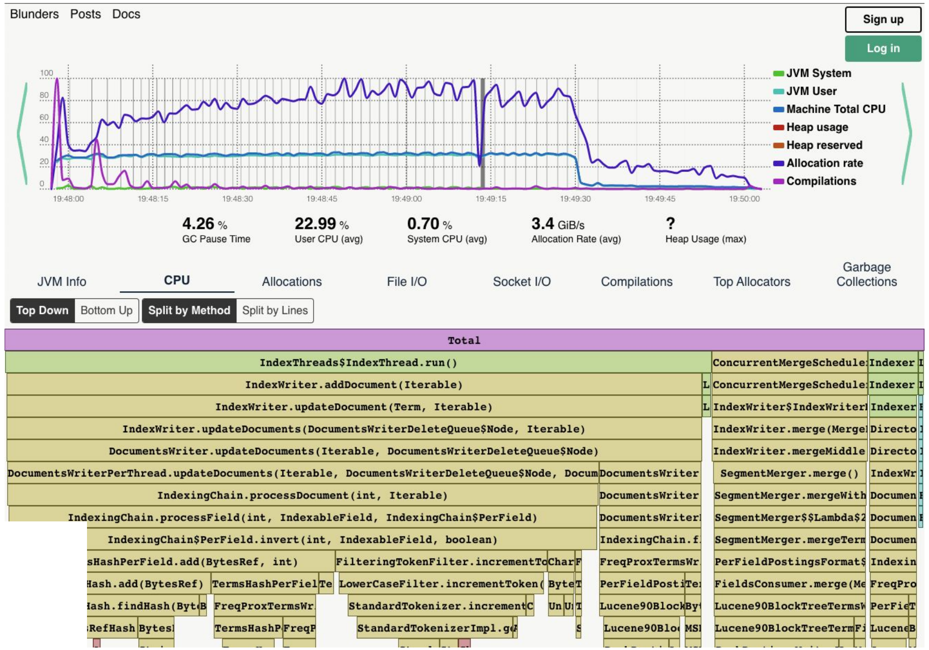
- Open source ASL2: luceneutil
- Open corpora: Wikipedia, OpenStreetMaps, NYC Taxi Rides, europarl
- Python to script the benchmark, Java to run each iteration
- Multiple threads run a continuous mix of diverse search tasks
- Focus on single-thread time to run each query
- Also: stored fields, faceting, sparse documents, geo-spatial, text analysis

Nightly benchmarks

- Runs same set of (many) tasks / indexing every night
- Takes ~10 hours each night, on a fast machine ("beast3")
- Tests latest mainline code, upgrade JDKs/OS frequently
- Creates interactive charts like Indexing and TermQuery
- Validates correctness ... regolding

Profiling

“The charts show you if something is fast or slow, not why” – Adrien Grand



Blunders Posts Docs
Sign up
Log in
JVM System
JVM User
Machine Total CPU
Heap usage
Heap reserved
Allocation rate
Compilations
4.26 % GC Pause Time
22.99 % User CPU (avg)
0.70 % System CPU (avg)
3.4 GiB/s Allocation Rate (avg)
? Heap Usage (max)
JVM Info
CPU
Allocations
File I/O
Socket I/O
Compilations
Top Allocators
Garbage Collections
Top Down
Bottom Up
Split by Method
Split by Lines
Total
IndexThreads$IndexThread.run()
IndexWriter.addDocument(Iterable)
IndexWriter.updateDocument(Term, Iterable)
IndexWriter.updateDocuments(DocumentsWriterDeleteQueue$Node, Iterable)
DocumentsWriter.updateDocuments(Iterable, DocumentsWriterDeleteQueue$Node)
DocumentsWriterPerThread.updateDocuments(Iterable, DocumentsWriterDeleteQueue$Node, Docum
IndexingChain.processDocument(int, Iterable)
IndexingChain.processField(int, IndexableField, IndexingChain$PerField)
IndexingChain$PerField.invert(int, IndexableField, boolean)
ConcurrentMergeSchedule
IndexWriter.merge(Merge
IndexWriter.mergeMiddle
SegmentMerger.merge()
SegmentMerger.mergeWith
SegmentMerger.mergeTerm

```
Profiler for cpu:
PROFILE SUMMARY from 561699 events (total: 561699)
  tests.profile.mode=cpu
  tests.profile.count=50
  tests.profile.stacksize=1
  tests.profile.linenumbers=false
PERCENT       CPU SAMPLES   STACK
10.08%        56630         org.apache.lucene.util.BytesRefHash#equals()
9.63%         54088         org.apache.lucene.index.TermsHashPerField#writeByte()
5.35%         30071         org.apache.lucene.analysis.standard.StandardTokenizerImpl#getNextToken()
4.22%         23706         org.apache.lucene.util.StringHelper#murmurhash3_x86_32()
3.88%         21797         java.io.FileOutputStream#write()
3.82%         21473         org.apache.lucene.analysis.standard.StandardTokenizer#incrementToken()
3.20%         17981         org.apache.lucene.index.TermsHashPerField#writeVInt()
2.99%         16770         org.apache.lucene.index.IndexingChain$PerField#invert()
2.85%         16031         sun.nio.ch.FileDispatcherImpl#write0()
2.72%         15290         org.apache.lucene.index.MappedMultiFields$MappedMultiTermsEnum#postings()
2.72%         15278         java.lang.Character#codePointAt()
2.41%         13565         org.apache.lucene.util.BytesRefHash#findHash()
2.22%         12480         java.lang.invoke.VarForm#getMemberName()
1.95%         10927         org.apache.lucene.analysis.standard.StandardTokenizerImpl#zzRefill()
```

Blunders.io integration

Outline

Quick introduction to Apache Lucene

Overview of our benchmark tooling

Battle scars!

Signal vs noise

- Benchmarks are noisy thanks to GC, Hotspot compilation (plus OS, hardware)
- Discard warmup/outliers, run many iterations (tasks and separate JVMs)
- Added confidence (p-values) recently
- Two schools of thought
    - Try JVM flags like -Xbatch -Xint -XX:-TieredCompilation to reduce noise
    - Run at JVM defaults to match production (noise and all) and run more iterations
- -XX:+PrintCompilation -verbose:gc are helpful
- Noise over time stands out (example)

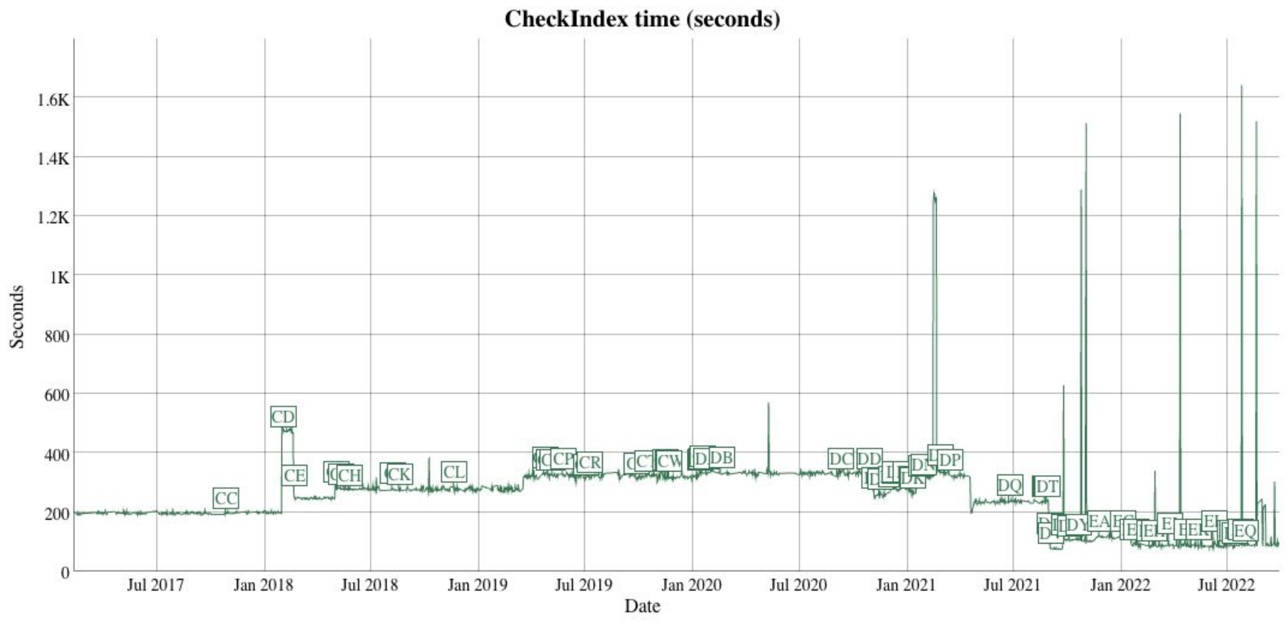
Deterministic Lucene index?

- A Lucene index has multiple segments…
- … but that impacts search performance and adds noise
- Solution?: single threaded indexing, but…
- … that’s slow (~6 hours)!
- Better solution: `IndexRearranger` (in progress)
- But not realistic? How to reflect improvements in merging?

Can we trust our benchmarks?

- Are results reproducible? Across different environments, developers, servers?
- Testing realistic workloads?
- Lurking bugs in the benchmarking tools?
- Is the nightly hardware too different from “normal” servers?
- Trust is *vital* – quickly address issues that erode trust!

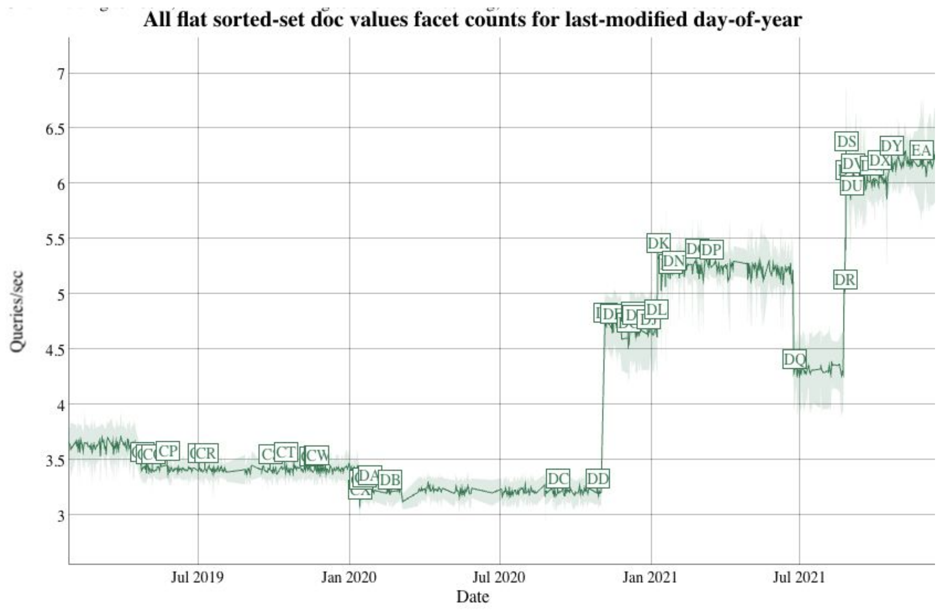
The WTF



CheckIndex time

The WTF

- Time consuming to root cause!
- Often you notice it days/weeks later
- You may discover other WTFs ("crabs")
- We need auto-WTF alarms
- Things may get even better after fixing: [example]



Too many changes at once!

- Sometimes nightly benchmarks are down for some time
- Sometimes we do a JDK upgrade, OS / Kernel upgrade, lots of Lucene changes land
- We push changes to the benchmarks themselves
- Suddenly benchmark breaks and we have to isolate
- Hardware, OS, JDK, benchmark tooling, Lucene can all change!

Benchmarks should not block good changes

- Benchmark is only one signal!
- If a change is a good simplification but makes things a bit slower, fine
- If a change makes slow queries faster, and fast queries a bit slower, fine
- A new feature should not have to satisfy any benchmarks before pushing
- It’s great to add new benchmarks for new features, but should not block the feature

When benchmarks catch bugs

- Sometimes nightly benchmark fails due to a Lucene bug
- Scary! It means our unit tests lack coverage…
- All hands on deck
- Example:

```
EXC: <vector:knn:<golf>[-0.07267512,...]>
java.io.EOFException: seek past EOF: MMapIndexInput(path="/index/lucene_bench_2021-01-25/index/_32.vec") [slice=vector-data]
	at org.apache.lucene.store.ByteBufferIndexInput.seek(ByteBufferIndexInput.java:255)
	at org.apache.lucene.store.ByteBufferIndexInput$MultiBufferImpl.seek(ByteBufferIndexInput.java:575)
	at org.apache.lucene.codecs.lucene90.Lucene90VectorReader$OffHeapVectorValues.vectorValue(Lucene90VectorReader.java:432)
	at org.apache.lucene.util.hnsw.HnswGraph.search(HnswGraph.java:118)
```

"OK, I was also able to reproduce this EOFException. It only seems to occur for the largest index, and I note that the file being read is > 2GB, so my guess is we have an integer/long problem somewhere." – Mike Sokolov

New benchmarks are born!

- When a performance regression escapes release and nightly benchmarks
- We dig to root cause and fix it…
- … and (hopefully) add a new benchmark case to test it going forwards
- Example: #10866
  - Origin story for dedicated stored fields benchmark
  - … which then uncovered another (merging) performance issue!
- #203 (CombinedFieldsQuery) merged two days ago
- Faceting benchmarks have also improved recently

More lessons/challenges

- Hard work to keep benchmarks working – APIs change, new build tooling upgrade OS and JDK, add coolant liquid, new features (e.g. KNN search)
- Hardware upgrade (three times now) causes misleading jumps across the board
- Benchmarks find exotic Lucene bugs
- A change in JDK's defaults can hurt Lucene performance (e.g. FuzzyQuery1)
- Top hits sometimes break!

Limitations

- Benchmark code is scratchy and smelly and has no unit tests!
- Missing red-line QPS (capacity)
- Missing long-pole latencies (no open loop tests: coordinated omission bug)
- We lack coverage on some Lucene features (highlighting, joins)
- No micro-benchmarks (use JMH?)
- Every PR should be tested, quickly – GitHub actions?

Patches/PRs Welcome!

The End