Apache Lucene™ on Amazon.com

Amazon

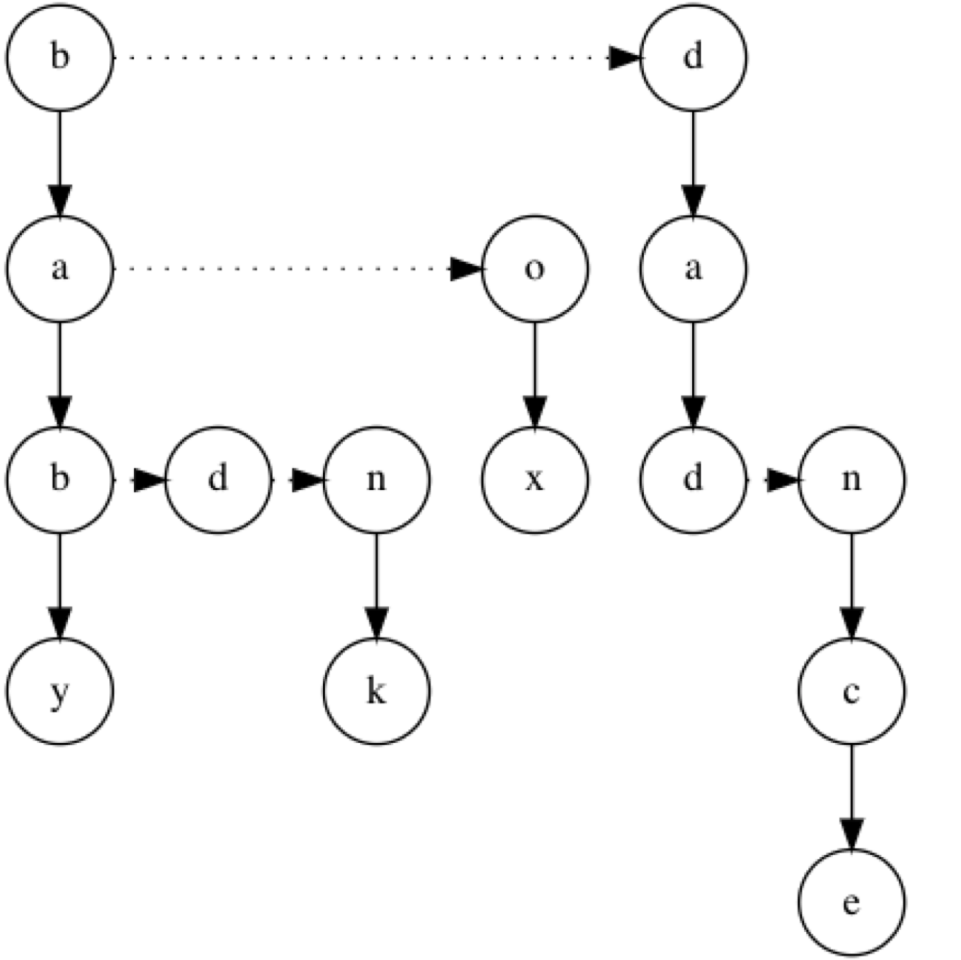
06.17.2019

Who are we?

- Mike McCandless
  - Long-time Lucene committer, author Lucene in Action 2nd edition
- Mike Sokolov
  - Search Veteran, new Lucene/Solr committer
- Contributions from many teammates @Amazon in Boston, Palo Alto, San Francisco, Dublin, Tokyo, Seattle

Outline

- Overview
- Service architecture
- Performance measurement
- Analysis challenges
- Query optimizations
- Multiphase ranking
- Summary



Using Lucene for shopping on Amazon

Amazon has strong search requirements:

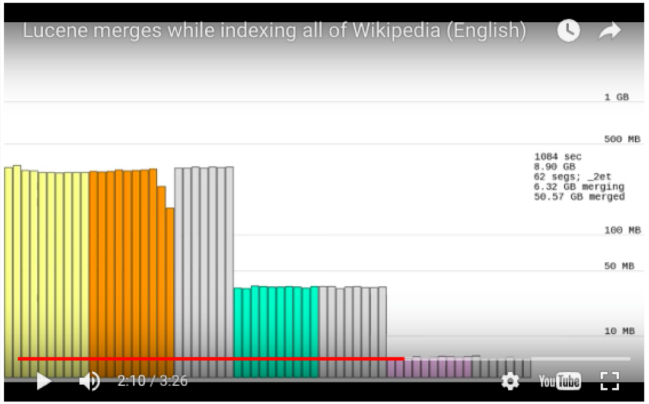
- High and peaky query rate
- Low latency bound
- Large, volatile catalog

Can Lucene handle these requirements?

Why Lucene?

- Lucene is a modern, mature, feature rich IR engine
- 20 years old!
- Apache open-source model, with generous license, works well
- Widely used inside and outside Amazon
- Active, passionate community is always innovating (e.g., maxscore scoring, Weak AND, Codec impacts in 8.0)
- Rich text analytics (full Unicode), modern scoring models, pluggable codecs

Lucene design

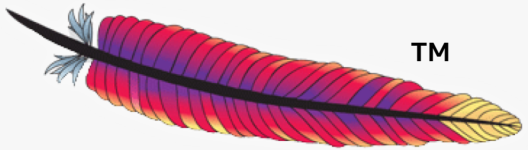


- 100% Java
- On-disk search index with small in-memory index structures
- Lucene can search very large indices with little RAM
- **Highly concurrent** indexing and searching
- Memory-mapped IO, rely on OS to cache hot pages
- **Segmented design** gives fast updates to a single index
- Near-real-time, transactional “point in time” search
- **Write-once design** allows for good value compression

Lucene features we are using

- Near-real-time segment replication
- Concurrent searching (“thread per segment per query”)
- Index time joins, static index sort, early termination
- Dimensional points for range filters and lightning deals
- Custom Collector, DoubleValuesSource, Query
- Custom term frequency for behavioral signals
- Taxonomy facets
- Multi-phase ranking
- Expressions

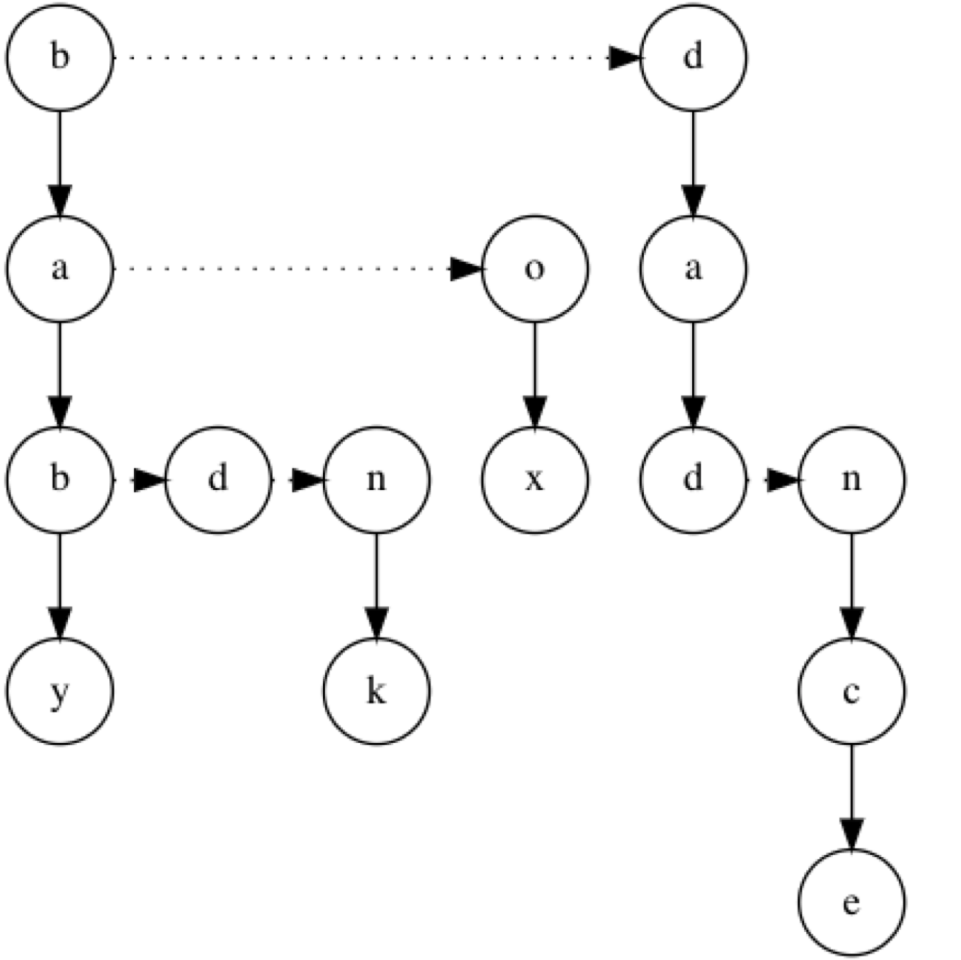
Open-source at Amazon



- Lucene is a strategic open-source project
- Developers are encouraged to interact with open-source community, push changes back, open issues, etc.
- Recent Lucene improvements:
  - Custom term frequencies
  - Concurrent indexing updates
  - Concurrent faceting
  - FST direct arc addressing
  - Off-heap FSTs

Outline

- Overview
- Service architecture
- Performance measurement
- Analysis challenges
- Query optimizations
- Multiphase ranking
- Summary

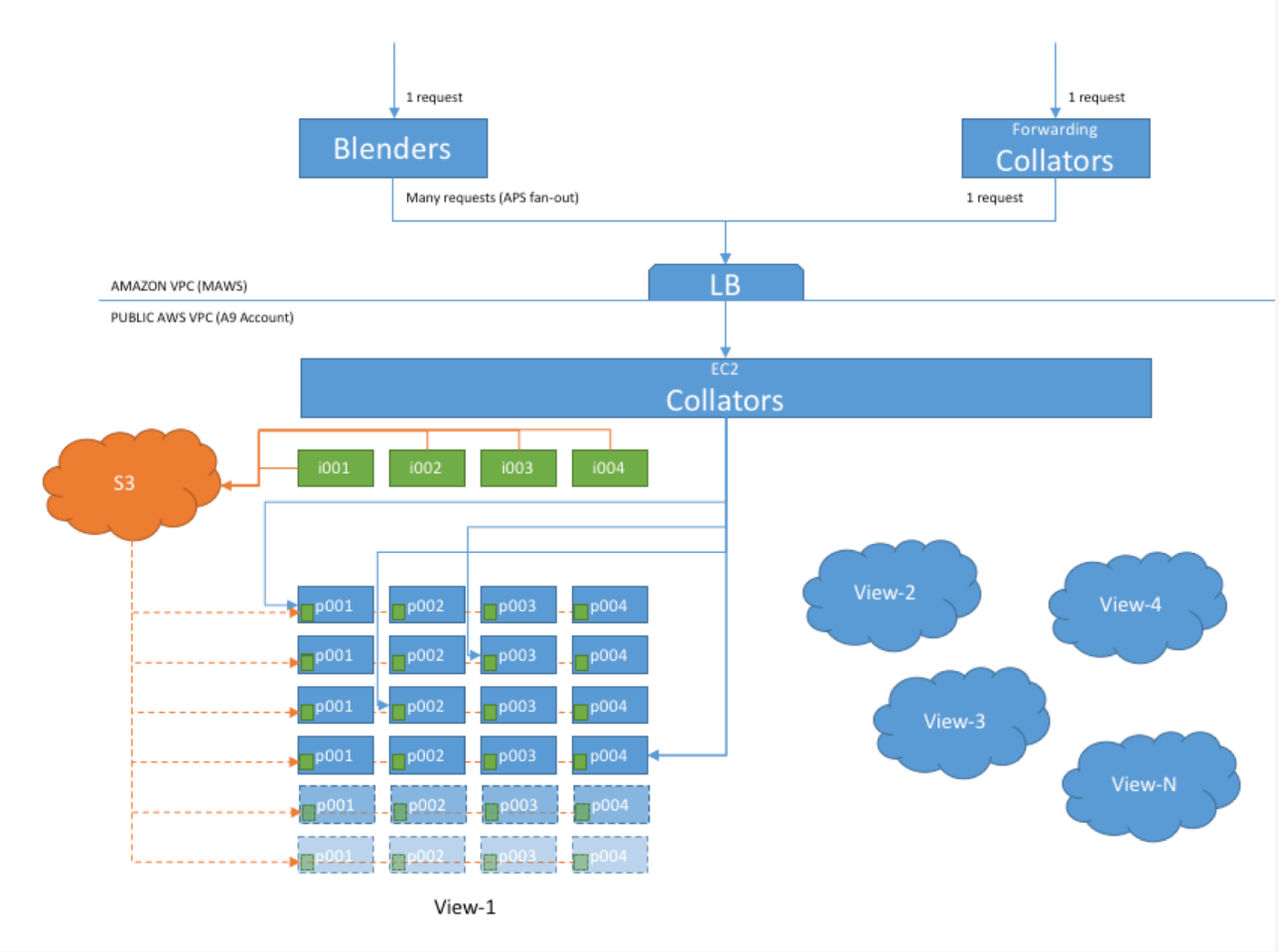


Near-real-time segment replication

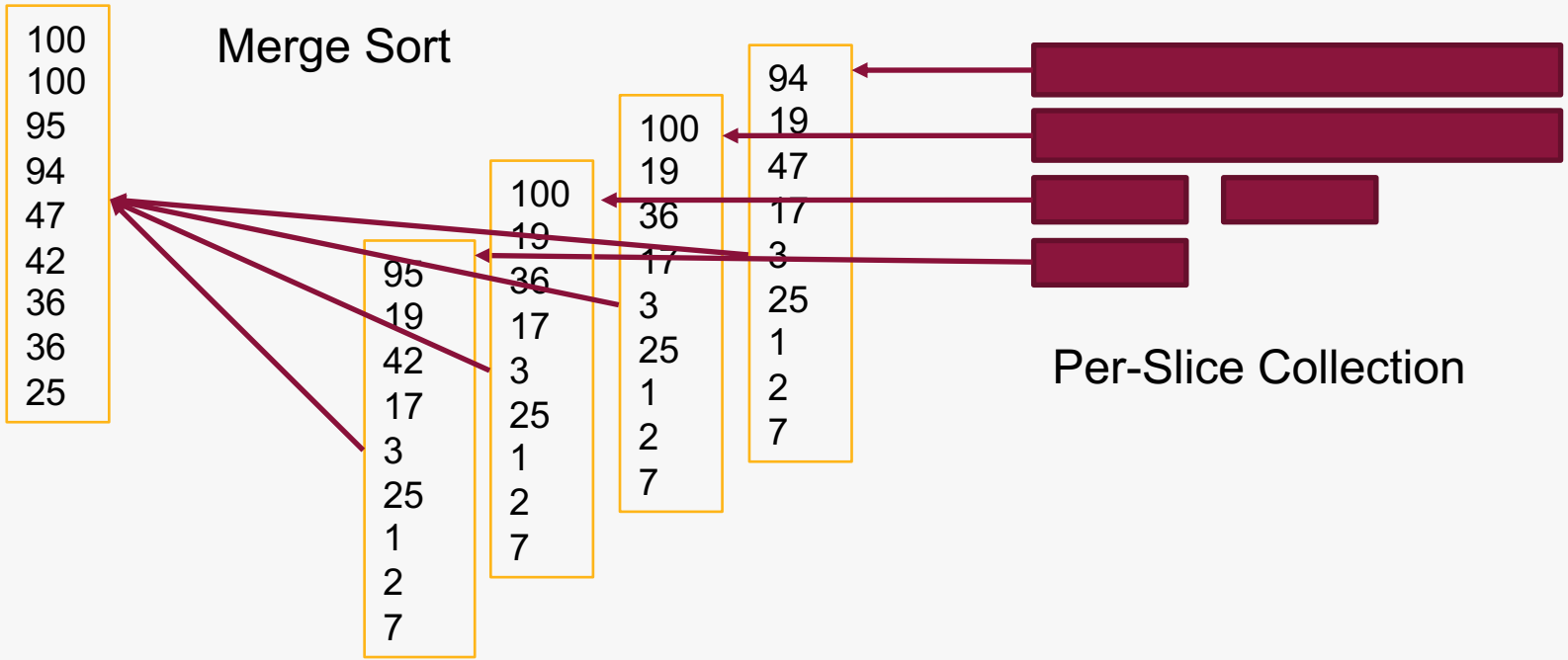
- Large indexes and low latency → sharded index
- High query throughput → replicated index
- Solr, Elasticsearch use document replication
- Lucene’s segments are a natural replication unit
- Index and merge each segment once
- Share segments using durable, highly-connected cloud storage
- External queue ensures no lost updates, consistency
- Preserve Lucene’s transactional semantics

Service architecture

- Build on AWS infrastructure
- Indexer, searcher nodes run inside ECS containers
- Catalog changes arrive via Kinesis queues and DynamoDB
- gRPC APIs trigger Lucene refresh, new near-real-time searcher every ~60 seconds
- Near-real-time index snapshots are saved in S3
- Index always re-built on each software deployment
- Service warmed using synthetic queries


1 request
Blenders
1 request
Forwarding
Collators
Many requests (APS fan-out)
1 request
LB
AMAZON VPC (MAWS)
PUBLIC AWS VPC (A9 Account)
EC2
Collators
S3
i001
i002
i003
i004
p001
p002
p003
p004
View-2
View-4
View-3
View-N
View-1

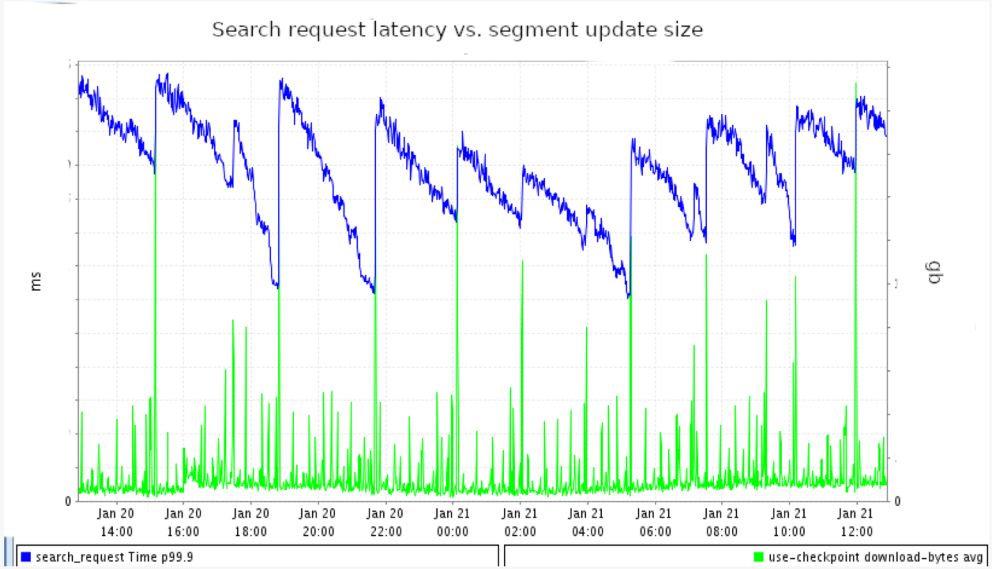
Searching a segmented index



Searching a segmented index concurrently

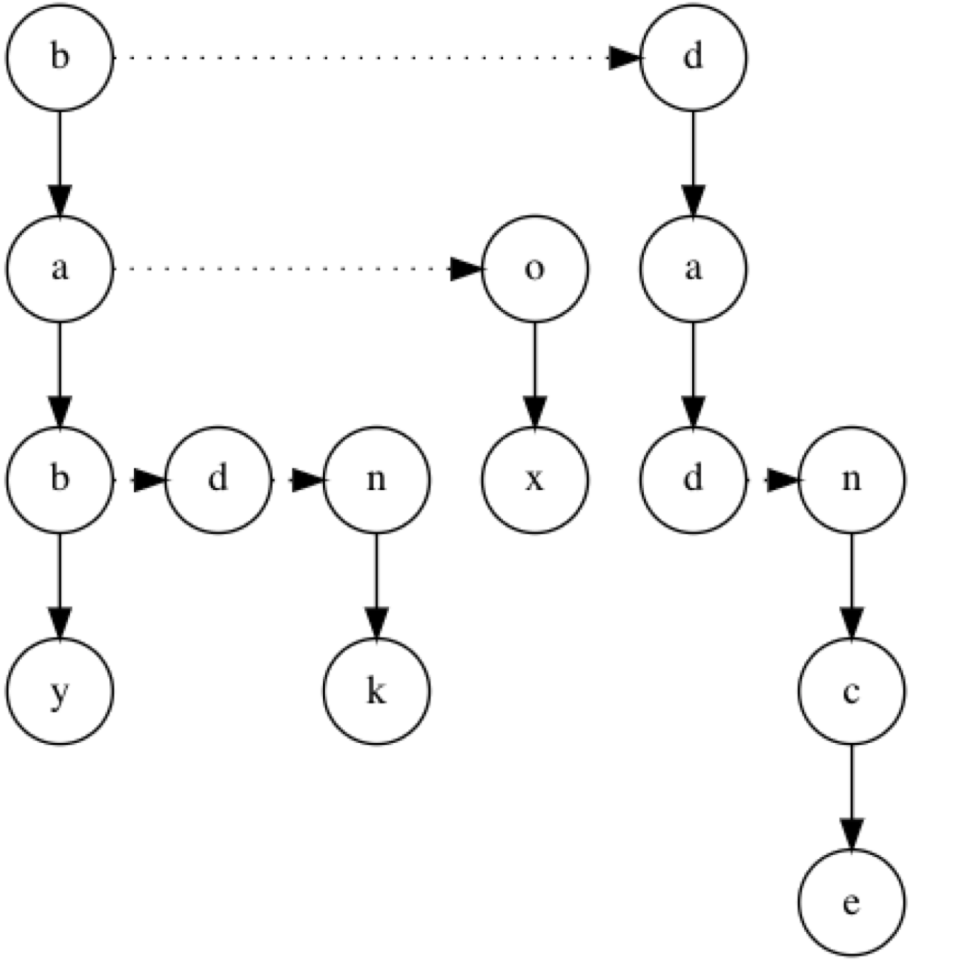
- Index is statically sorted by item “quality”
- Per-segment early termination
- Thread per segment per query
- Better long-pole query latencies, but worse red-line QPS
- Can we fall back to single threaded near red-line?
- Can we use multiple threads to search a large segment?

P99 query latency and segment replication



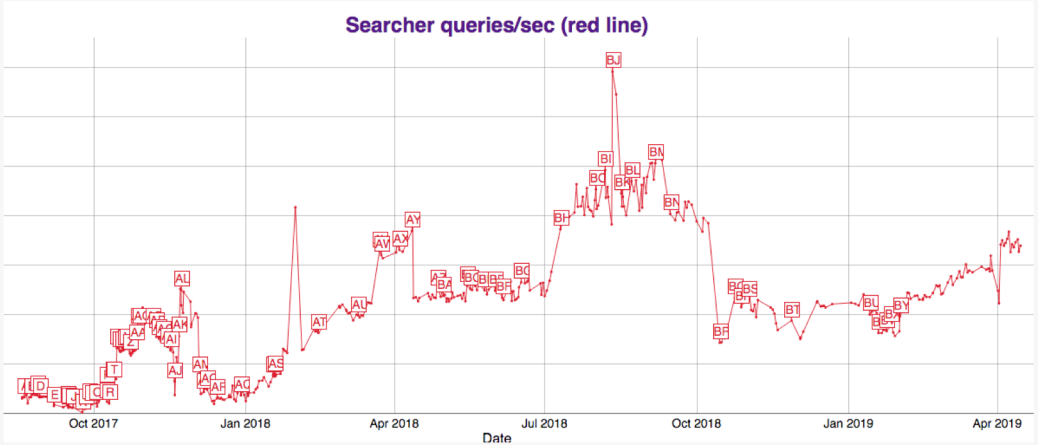
Outline

- Overview
- Service architecture
- Performance measurement
- Analysis challenges
- Query optimizations
- Multiphase ranking
- Summary



Internal nightly benchmarks

- Similar to Lucene's nightly benchmarks
- Track progress and catch accidental regressions
- Measure both functionality and performance metrics



Measuring performance

- Long-pole (P99+) query latency is a key metric
- Query latencies measured with open-loop client, Poisson arrival times
  - Avoids “coordinated omission” bug
  - Latency measured under what conditions?
- Red-line QPS measured with closed-loop client
- Goal: drive up red-line QPS while holding down latencies below red-line

Concurrent refresh

- Problem: Lucene “borrows” application indexing threads to provide concurrent refresh
- If application uses only one thread calling refresh(), that’s single threaded – common case?
- On highly concurrent hardware this is very slow
- Solution: use expert Lucene API to refresh concurrently (LUCENE-8700)

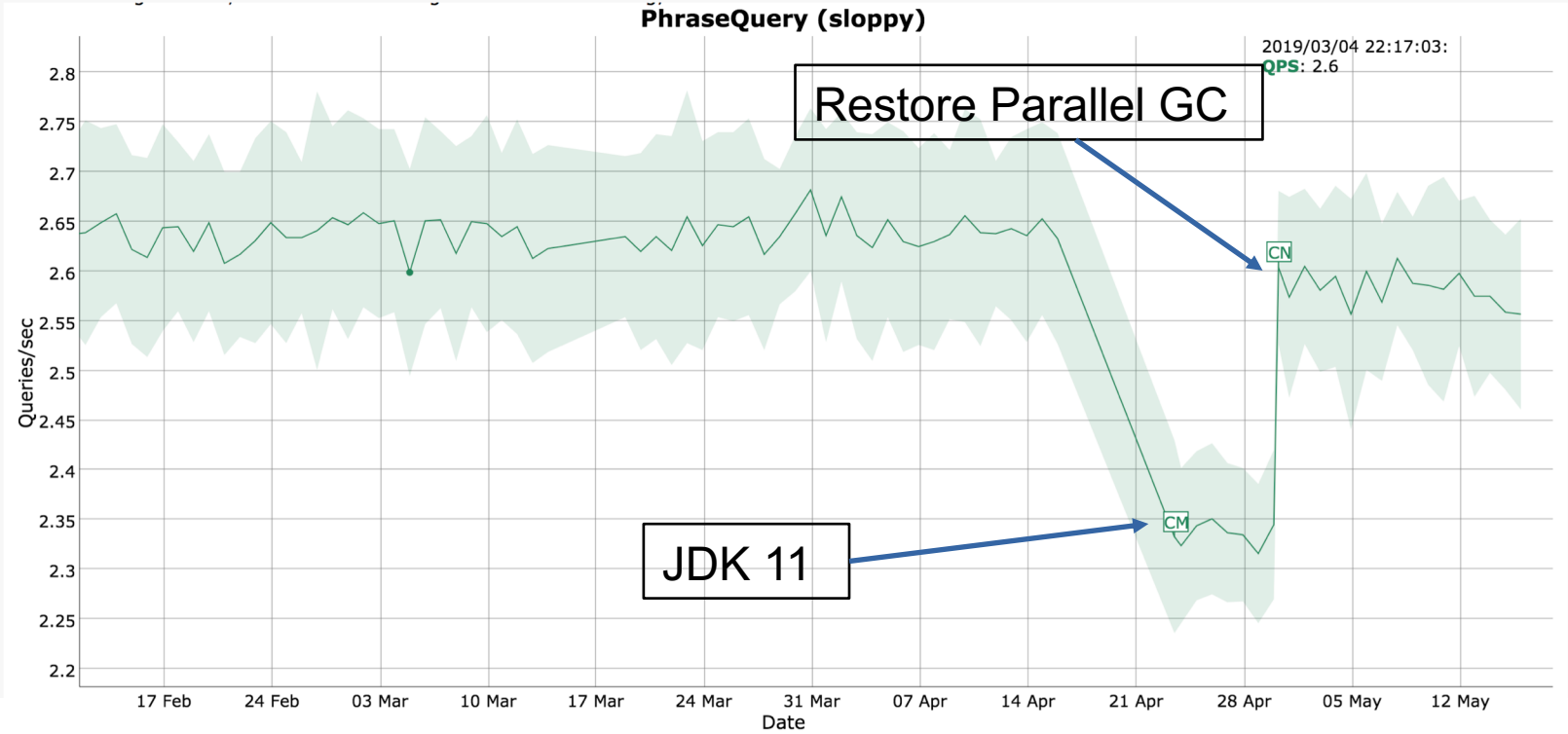
Gathering metrics using Lucene's abstractions

- Wrap DirectoryReader to count term lookups
- Wrap Directory, IndexInput to gather IO counters
- How many bytes does each query visit?
- How many times does each query lookup terms?
- Track per-query metrics in nightly benchmarks

Full garbage collection is bad!

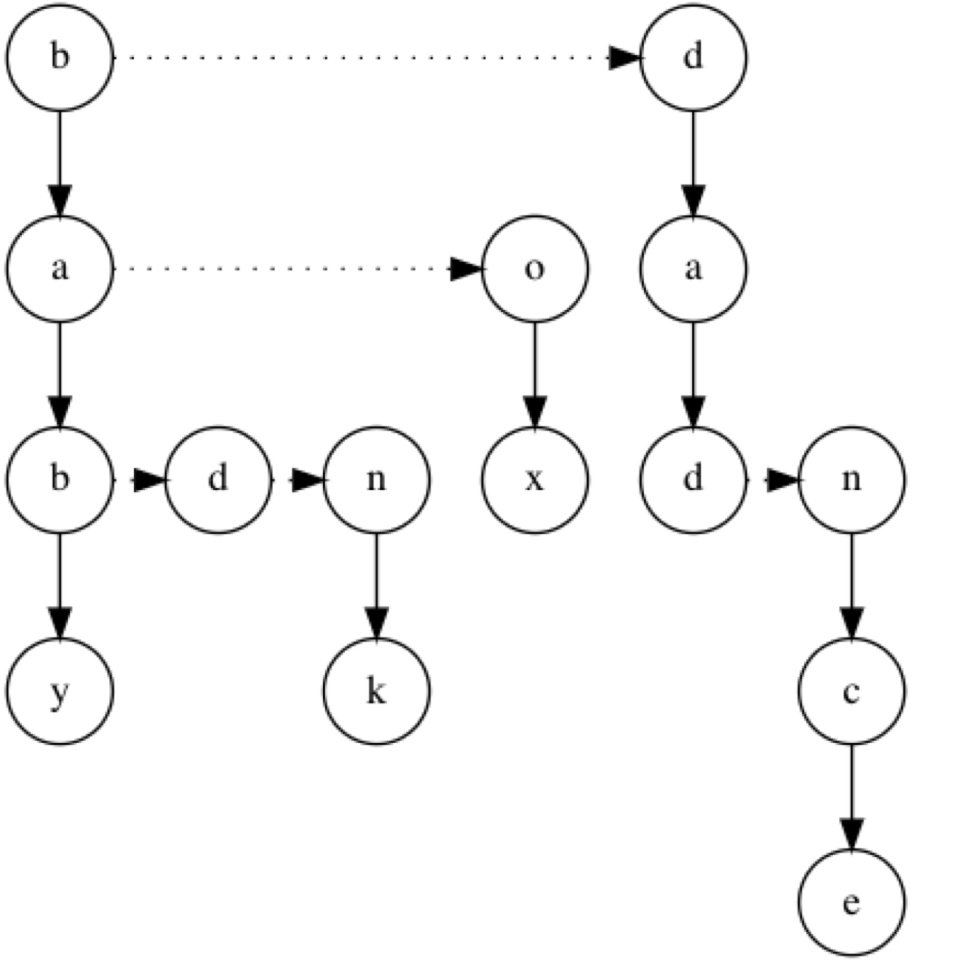
- We use JDK11's (deprecated) CMS garbage collector
- We don't trust G1GC yet
- Use Azul's jHiccup to measure real pauses
- We hit 8 second stop-the-world full GC pauses
  - Reduced heap usage
  - Increased heap size
  - Changed GC parameters (poached from Elasticsearch)
    - -XX:CMSInitiatingOccupancyFraction=75
    - -XX:+UseCMSInitiatingOccupancyOnly

Lucene nightly benchmarks



Outline

- Overview
- Service architecture
- Performance measurement
- Analysis challenges
- Query optimizations
- Multiphase ranking
- Summary



Context-sensitive analysis

- What does “plane” mean to you?

Context-sensitive analysis

An
Airplane?

Context-sensitive analysis

A bench plane?

Context-sensitive analysis

- Synonyms applied only during indexing
- We have a helpful synonym “plane” “airplane,” but we probably shouldn’t apply it to tools
- Lucene switches analysis per field
- We switch synonyms based on field values like product type, and other contextual information

Numbers are special

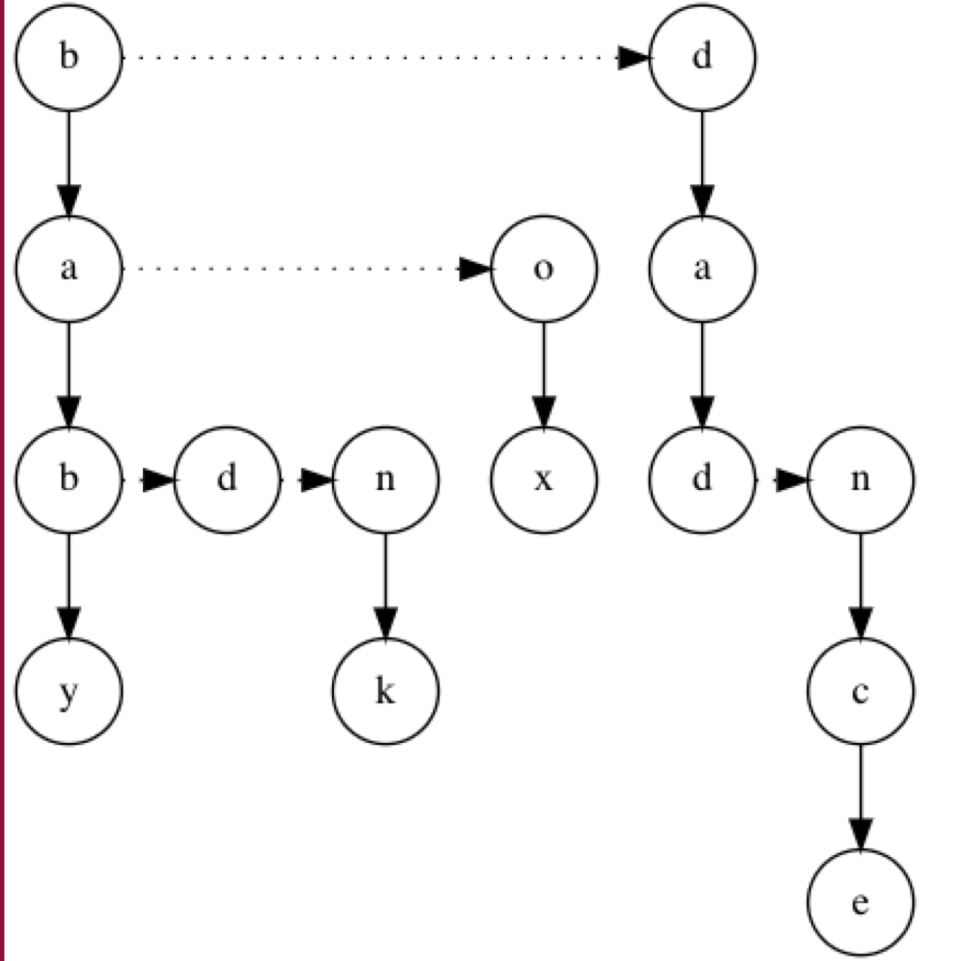
- “Toy for 3 year old” should match toys with text “for age 2-4 years”
- 1500 ml should match 1.5 liters.
- 1,100 == 1100 == 1.100 != 1/100 or 1:100
- It’s hard to handle these after StandardTokenizer!

WordDelimiterGraphFilter

- Splits on non-letter/number characters
  - Cannot accept a token graph
  - Messes up offsets
  - Many many options
- Useful with whitespace tokenization

Outline

- Overview
- Service architecture
- Performance measurement
- Analysis challenges
- Query optimizations
- Multiphase ranking
- Summary



Indexed queries

- Many queries share common sets of filters

```
+asin_or_proffer:asin +is_idq_suppressed:0
+is_campus_custom:0 +adult-product:0
```

- Let’s factor them out during indexing (like Percolate)
- … and searching, replacing with a single TermQuery clause

Factoring queries

- Factoring general Boolean expressions is hard!
- Luckily, our queries are mostly conjunctive
- FP-growth algorithm works well
- Simplify by handling one level of nesting

Results

- +30% red-line QPS!
- P99 latency 81ms -> 54ms

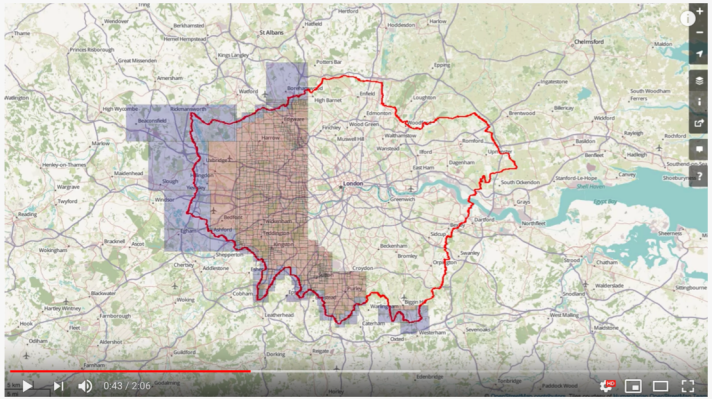
Indexing tuples

- This is a similar idea, but for full text
- Index common pairs of words (tuples)

casa_iphone: 20535
iphone_plus: 10297
dress_woman: 7956
shoe_woman: 7497
casa_galaxy: 5175
galaxy_samsung: 4912
led_light: 4854
day_valentine: 4840

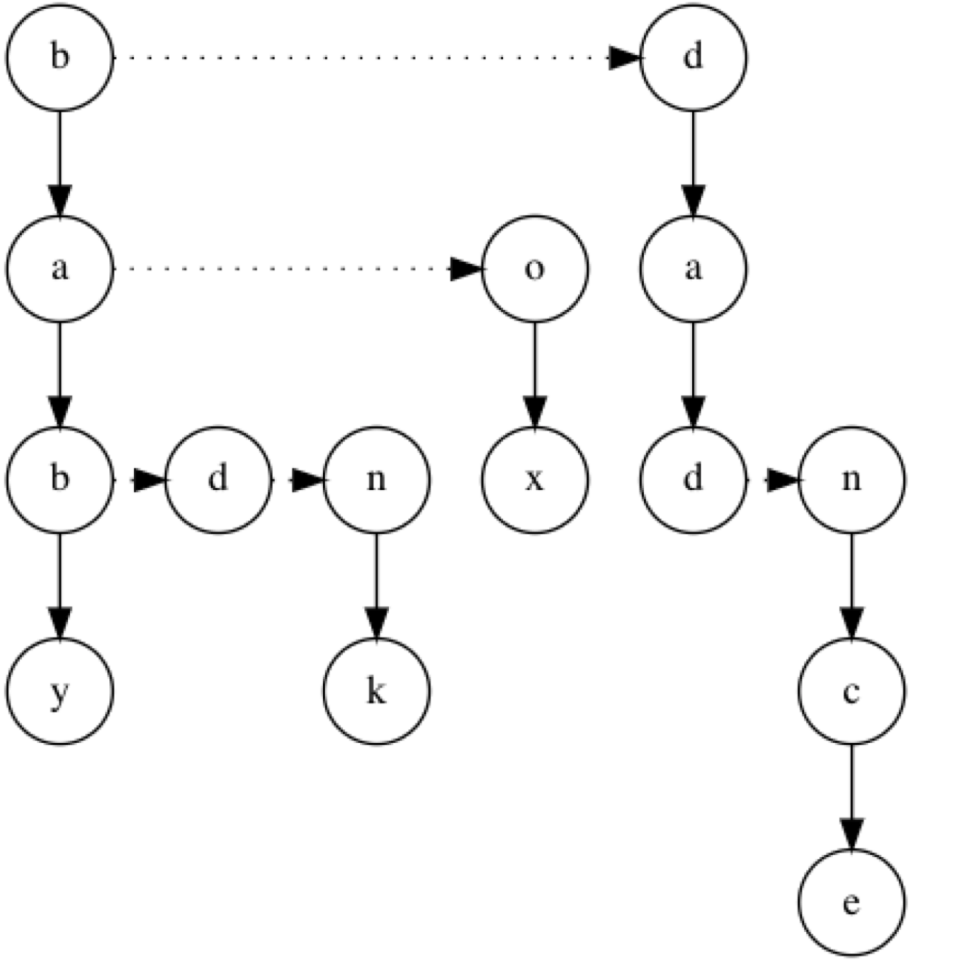
Lightning deals using dimensional points

- Each lightning deal has a unique name, and start/end time range
- Each product can have multiple deals
- Very time sensitive – e.g. on Prime Day 2019
- Custom 3D shape and query



Outline

- Overview
- Service architecture
- Performance measurement
- Analysis challenges
- Query optimizations
- Multiphase ranking
- Summary

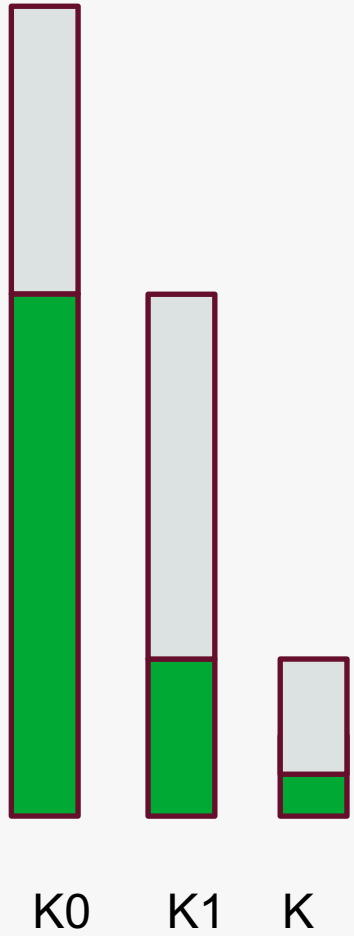


Ranking

- Machine-learned models using custom evaluator
- Multiple input signals
  - Custom term freqs for behavioral scores
  - Doc values fields for per-document signals
- Custom scoring functions as DoubleValuesSource
- Heavy use of Lucene expressions

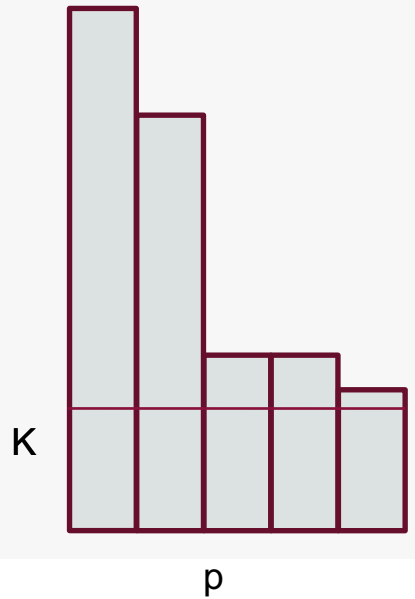
Multi-phase ranking

- Top K0 matching docs ordered by index rank
- Top K1 of K0 reordered with fast rank
- Top K of K1 with precise final rank
- Tunable tradeoff of speed/precision



Phase 0 concurrent collection

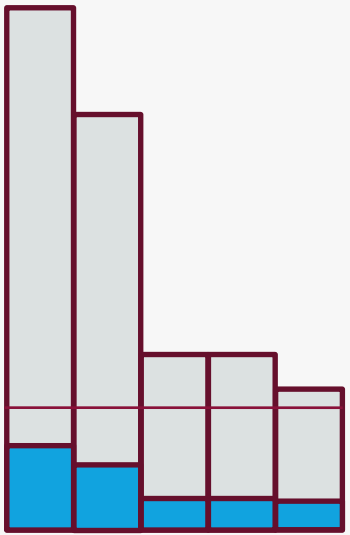
- Conservatively, collect K0 for each segment
- Guarantees same top K0 as sequential collection
- How likely is this worst case?
- For *random distribution* in p segments:
- (1/p)^K0; p ~ 20, and K0 ~1000
- (1/20)^1000 = not going to happen



Proportional collection

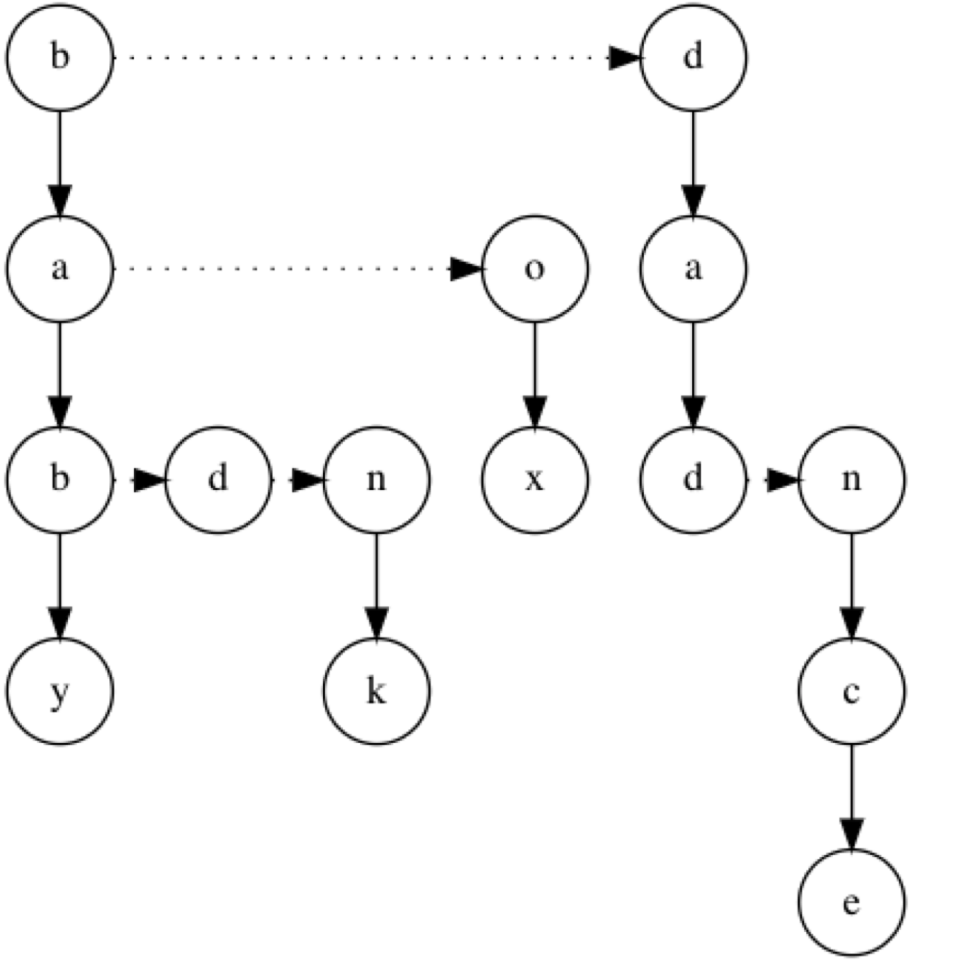
- Expected portion of top *n* in segment *k* is $n*p_k$
- LUCENE-8681
- Multinomial p.d.f gives probability (number of combinations) of a given document distribution

$$\frac{n!}{x_1! \cdots x_k!} p_1^{x_1} \times \cdots \times p_k^{x_k}$$



Outline

- Overview
- Service architecture
- Performance measurement
- Analysis challenges
- Query optimizations
- Multiphase ranking
- Summary



Summary

- Lucene works well for Amazon’s product search!
- Segment replication is efficient for deep clusters
- Thread per segment concurrency yields low latencies
- If you enjoy working on Lucene open source, and high scale, high impact software… come join us!

Amazon Search

Thank you

Amazon

QUESTIONS?

now … or come find us at our booth!