SearchWorkings

Lucene 4 - Next generation open source search

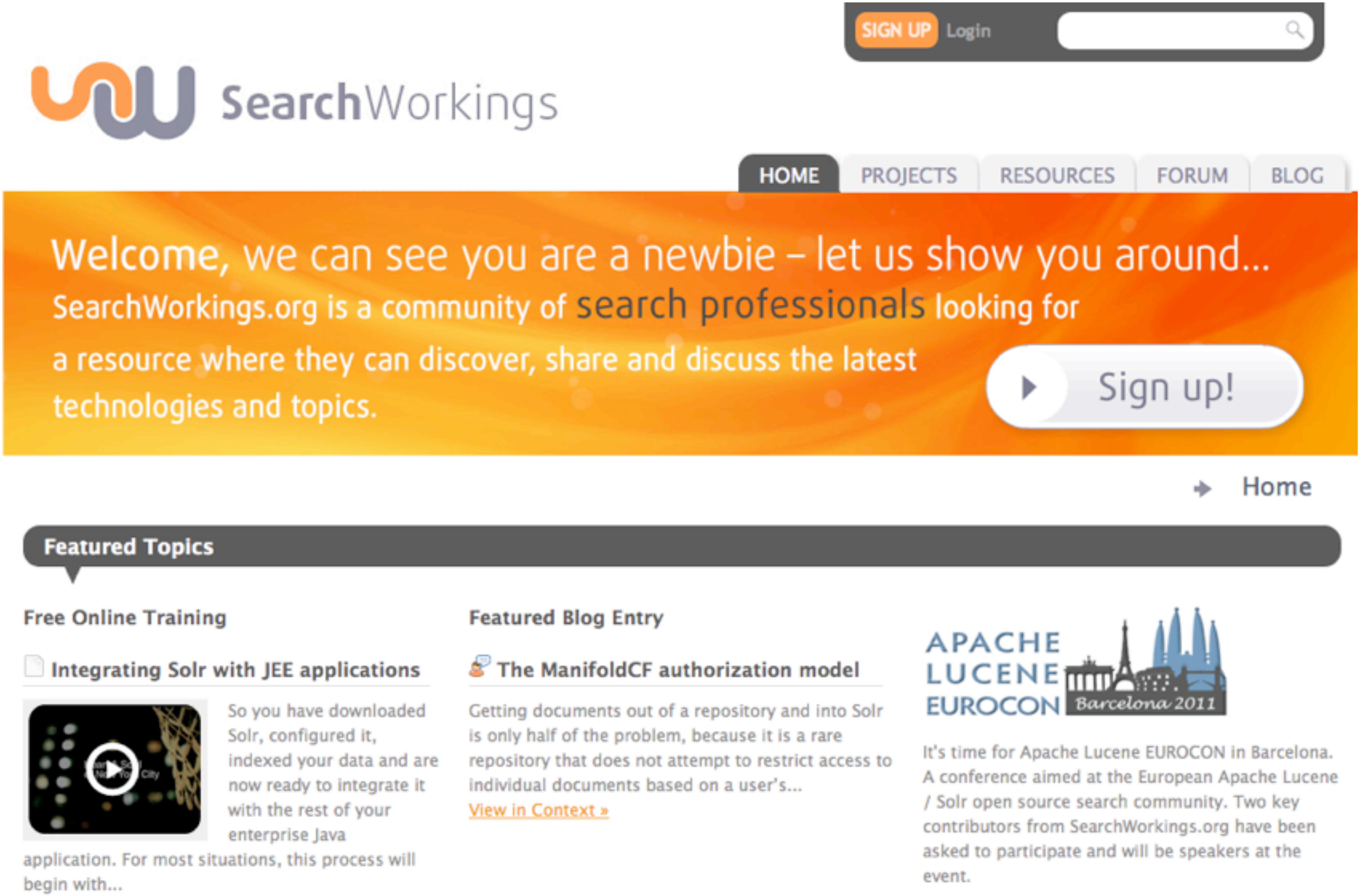
Simon Willnauer
Apache Lucene Core Committer & PMC Chair
simonw@apache.org / simon.willnauer@searchworkings.org

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 »

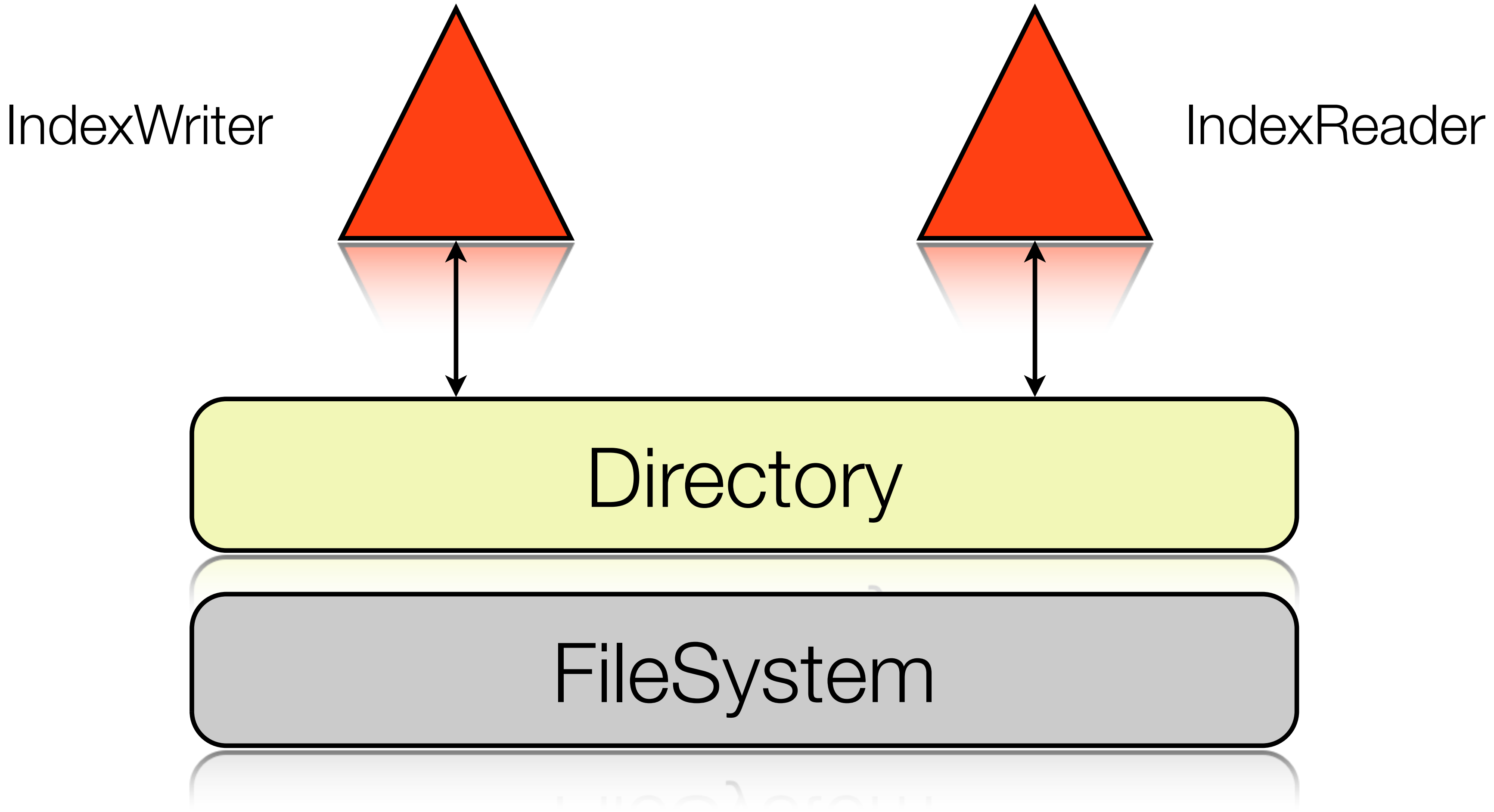
APACHE LUCENE EUROCON Barcelona 2011

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.

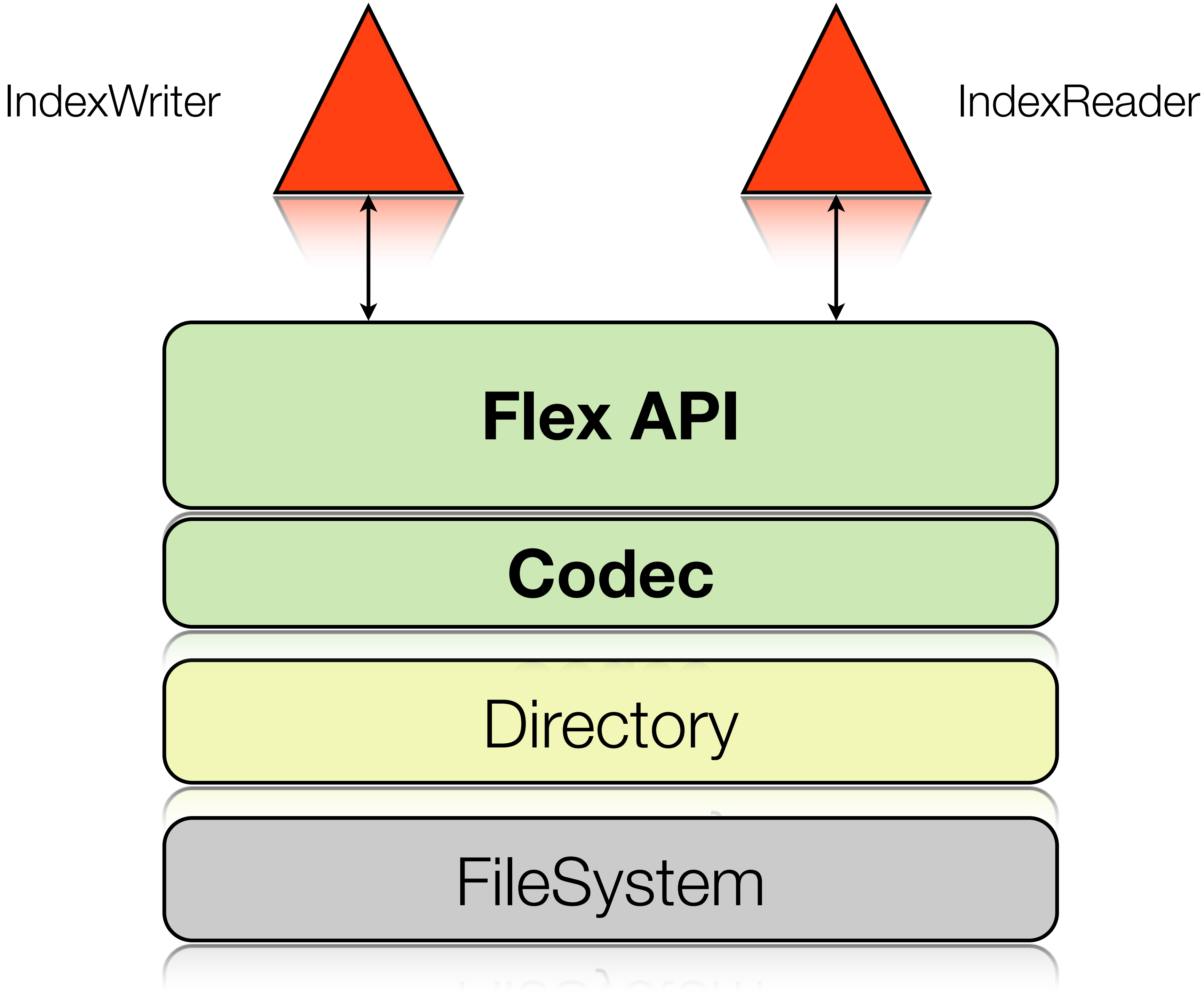
Agenda

- Flexible Indexing
- IndexDocValues
- DocumentsWriterPerThread (DWTP)
- Automaton Queries
- Random & Pending Improvements

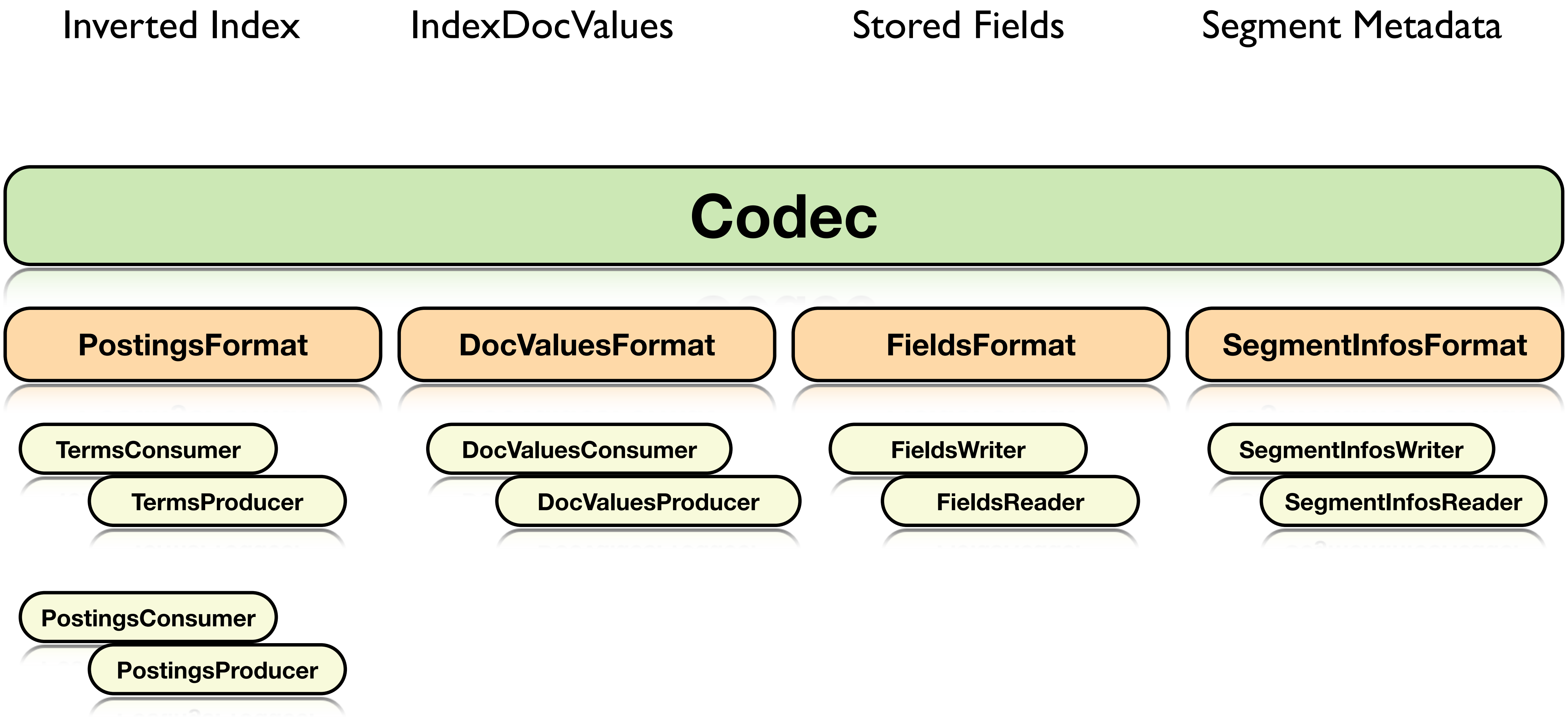
Architecture prior to Lucene 4.0



Architecture with Flexible Indexing



Lucene 4.0 Codec Layer



Inverted Index

IndexDocValues

Stored Fields

Segment Metadata

Codec

PostingsFormat

DocValuesFormat

FieldsFormat

SegmentInfosFormat

TermsConsumer

TermsProducer

DocValuesConsumer

DocValuesProducer

FieldsWriter

FieldsReader

SegmentInfosWriter

SegmentInfosReader

PostingsConsumer

PostingsProducer

Good news / Bad news

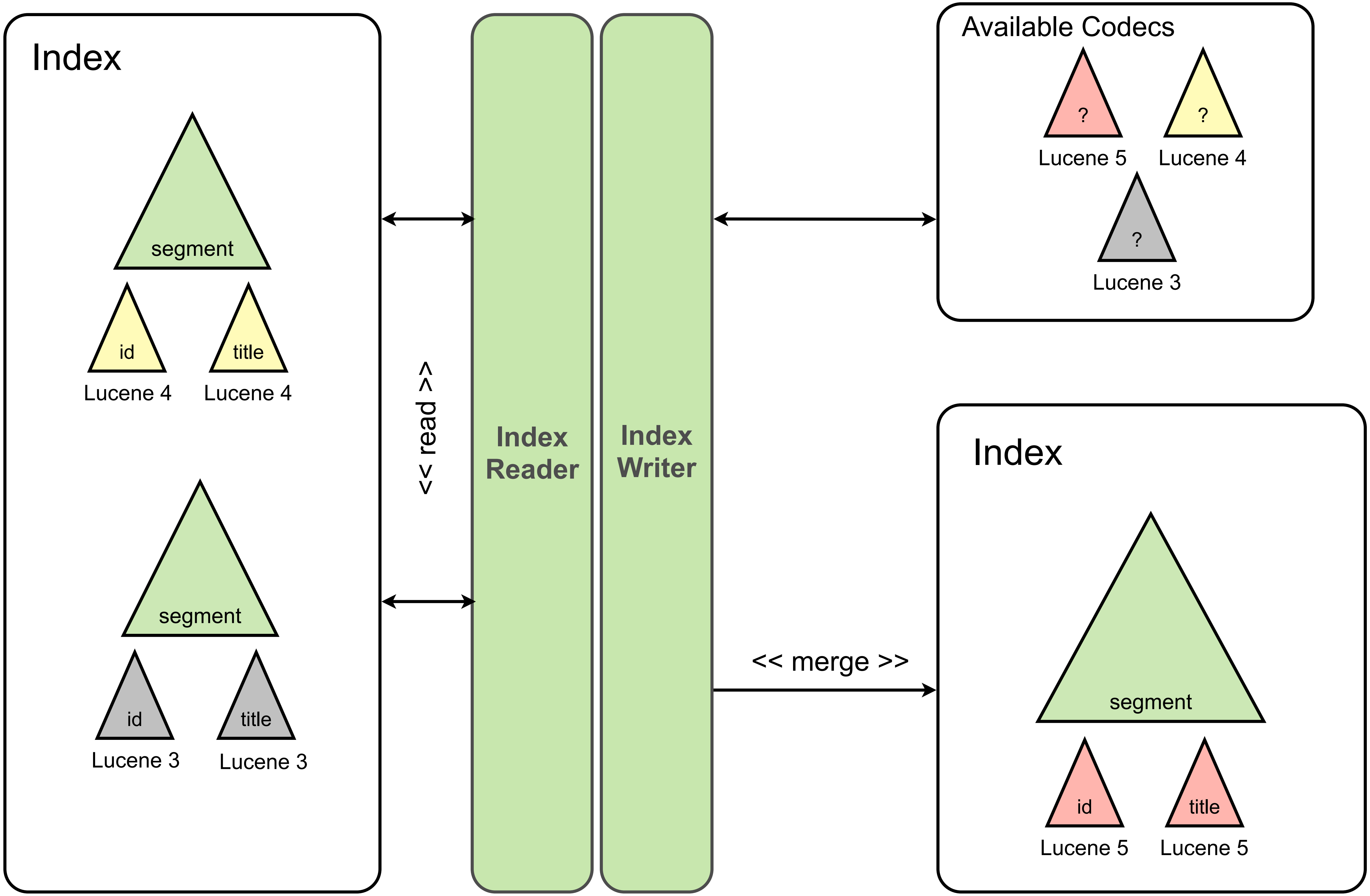
- 90% will never get in touch with this level of Lucene
- the remaining 10% might be researchers :)
- However - configuration options might be worth while

elasticsearch.

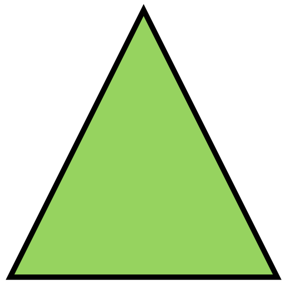
Apache Solr

- Why is this cool again?

For Backwards Compatibility you know?

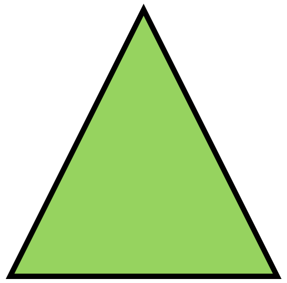


PostingsFormat Per Field



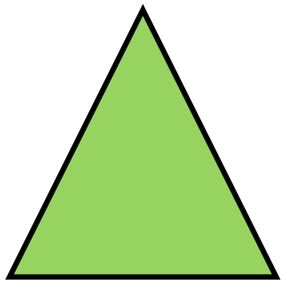
field: uid

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



field: spell

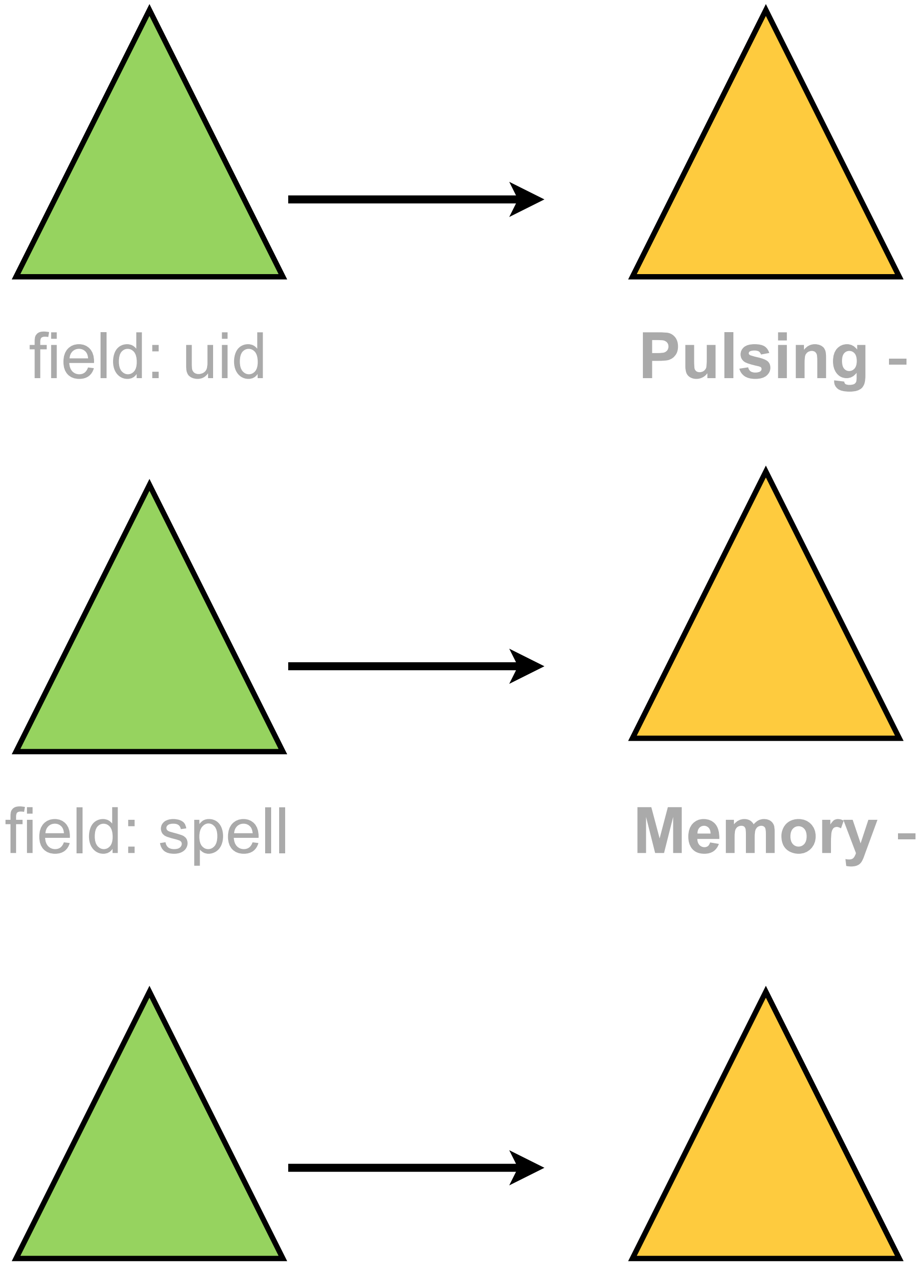
- large number of tokenized unique terms
- spelling correction - no posting list traversal
- large amount of key lookups



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

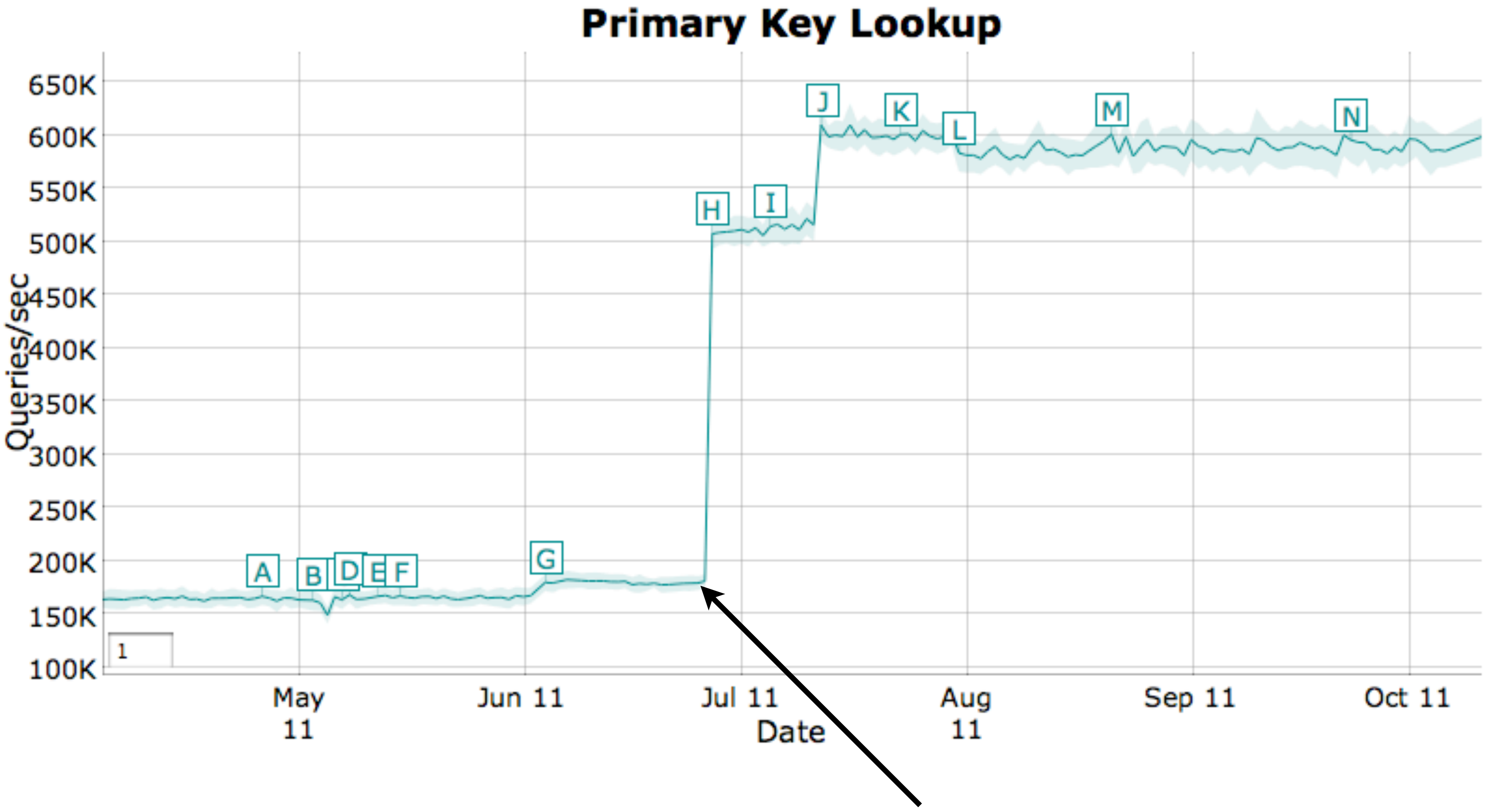
- loads terms & postings into RAM
- linear scanning vs. skipping
- in-mem FST usually very compact

field: body

Default - PostingsFormat

- very memory efficient
- terminates early for seekExact
- uses skipping for postings

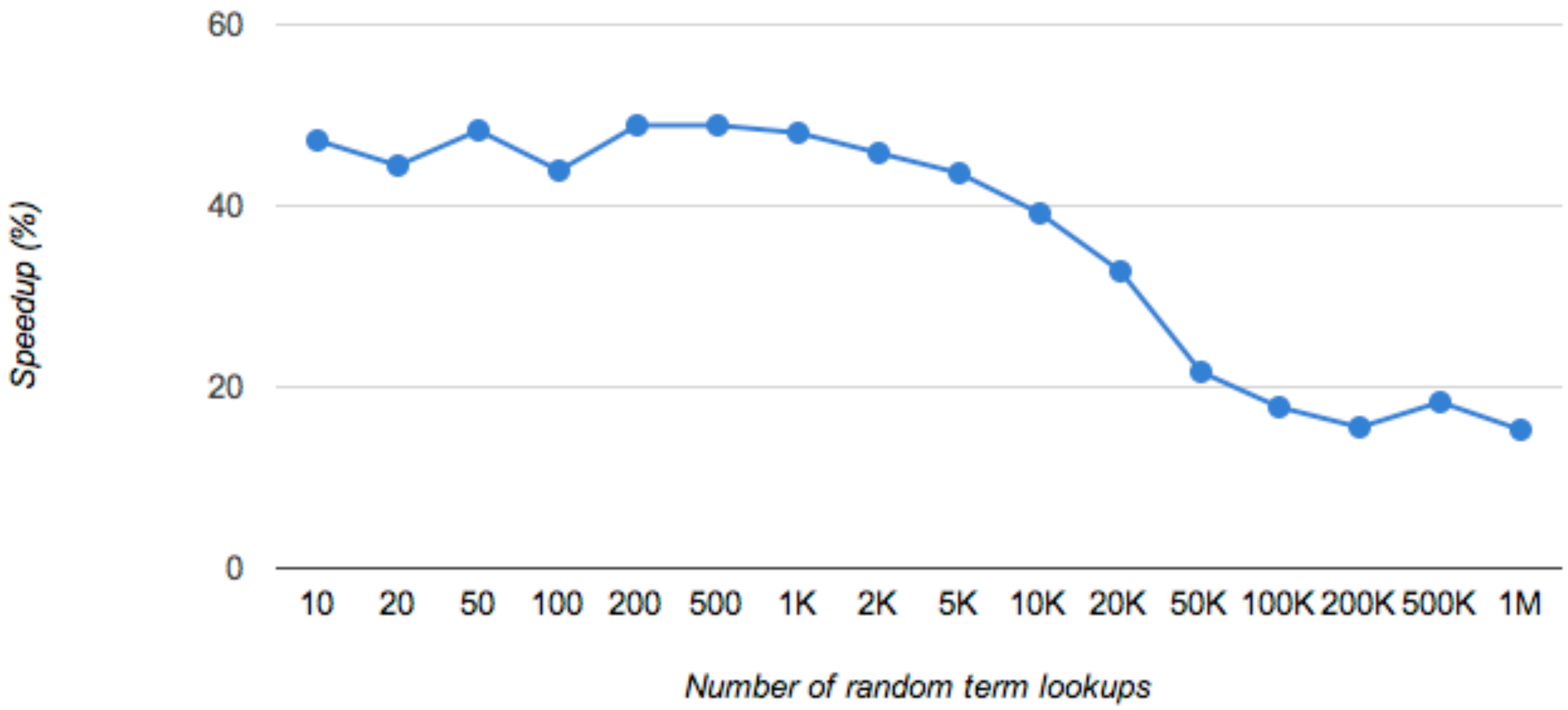
Using the right tool for the job..



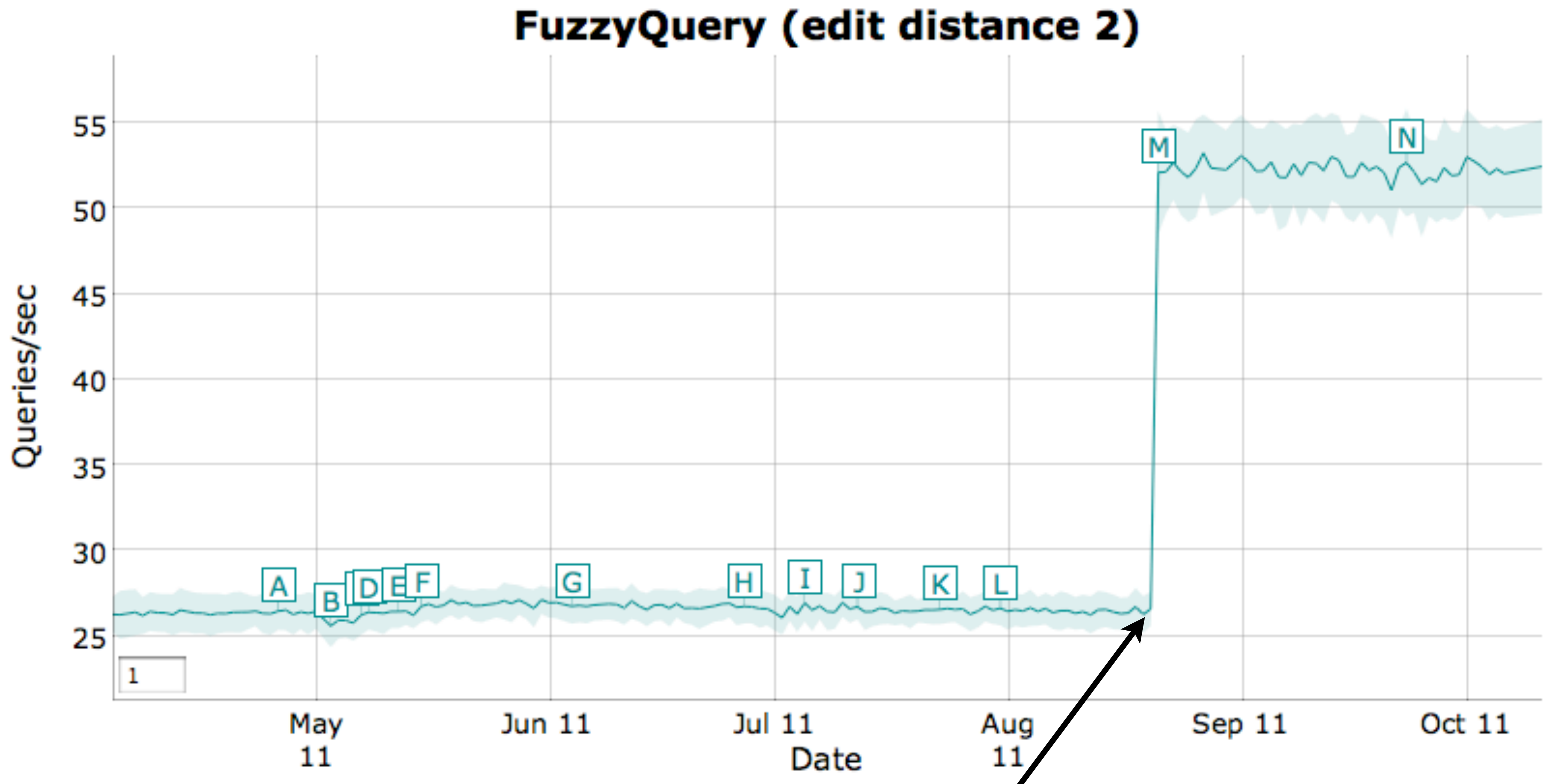
Switching to Memory PostingsFormat

Using the right tool for the job..

Speedup with Pulsing Codec



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**

IndexDocValues

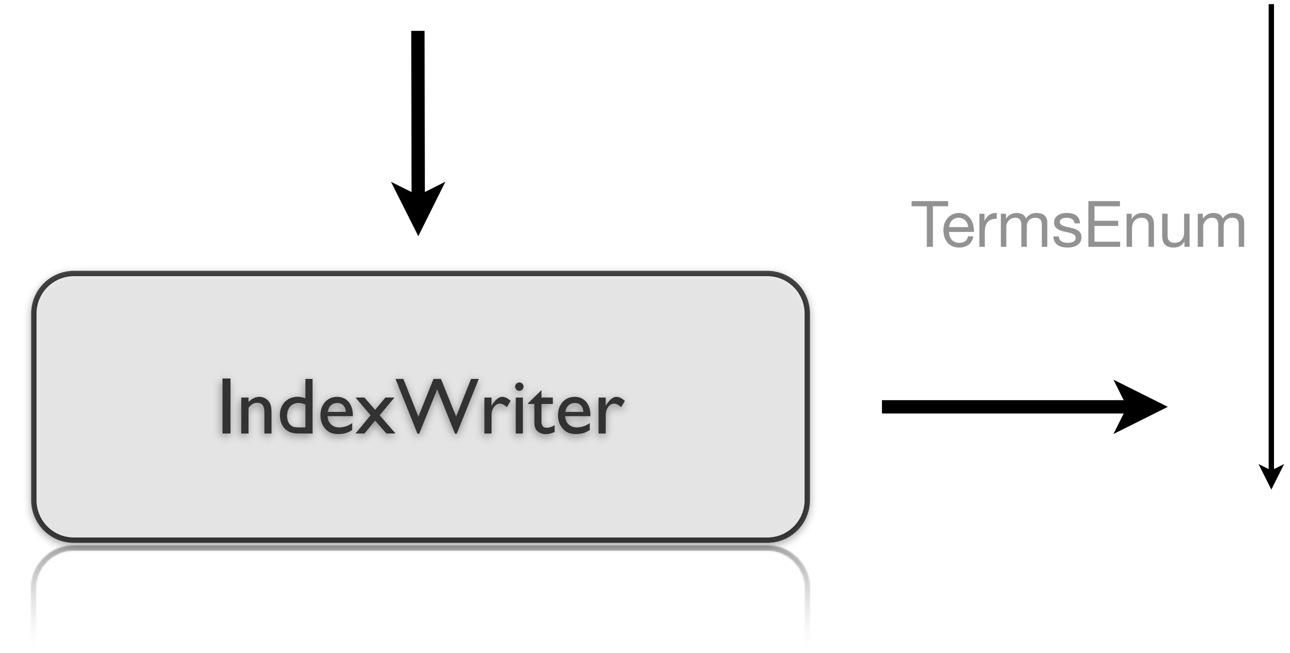
?

What is this all about? - Inverted Index

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

1	The old night keeper keeps the keep in the town
2	In the big old house in the big old gown.
3	The house in the town had the big old keep
4	Where the old night keeper never did sleep.
5	The night keeper keeps the keep in the night
6	And keeps in the dark and sleeps in the light.

Table with 6 documents



IndexWriter

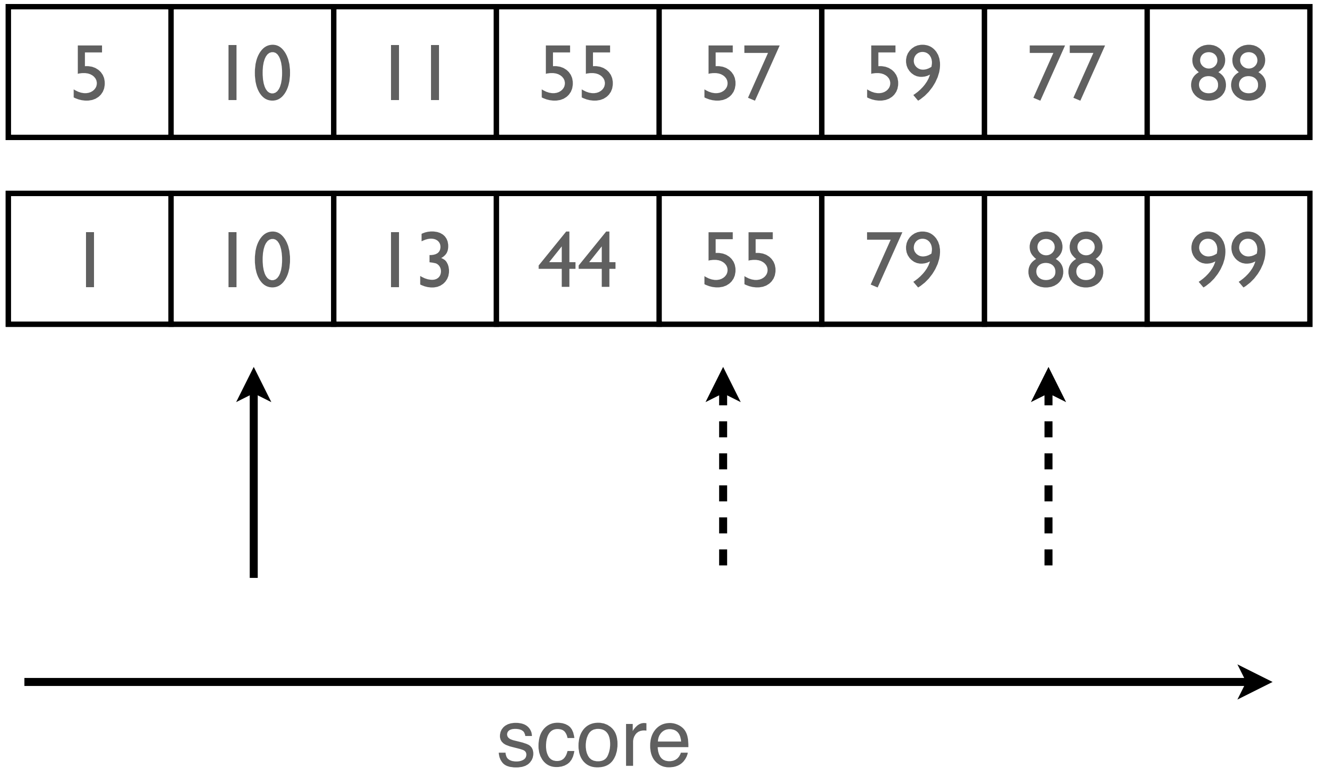
TermsEnum

term	freq	Posting list
and	1	6
big	2	2 3
dark	1	6
did	1	4
gown	1	2
had	1	3
house	2	2 3
in	5	<1> <2> <3> <5> <6>
keep	3	1 3 5
keeper	3	1 4 5
keeps	3	1 5 6
light	1	6
never	1	4
night	3	1 4 5
old	4	1 2 3 4
sleep	1	4
sleeps	1	6
the	6	<1> <2> <3> <4> <5> <6>
town	2	1 3
where	1	4

Intersecting posting lists

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

Posting Lists (document IDs)



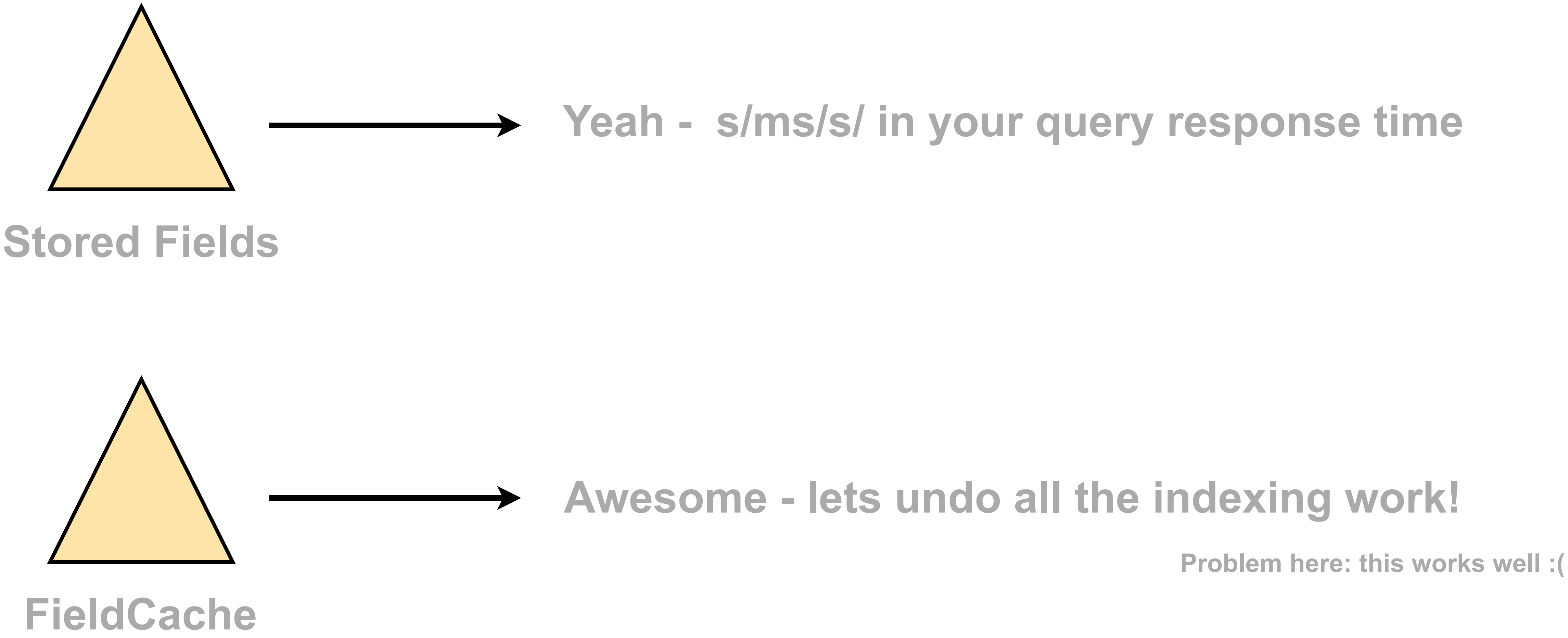
5	10	11	55	57	59	77	88
1	10	13	44	55	79	88	99

score

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

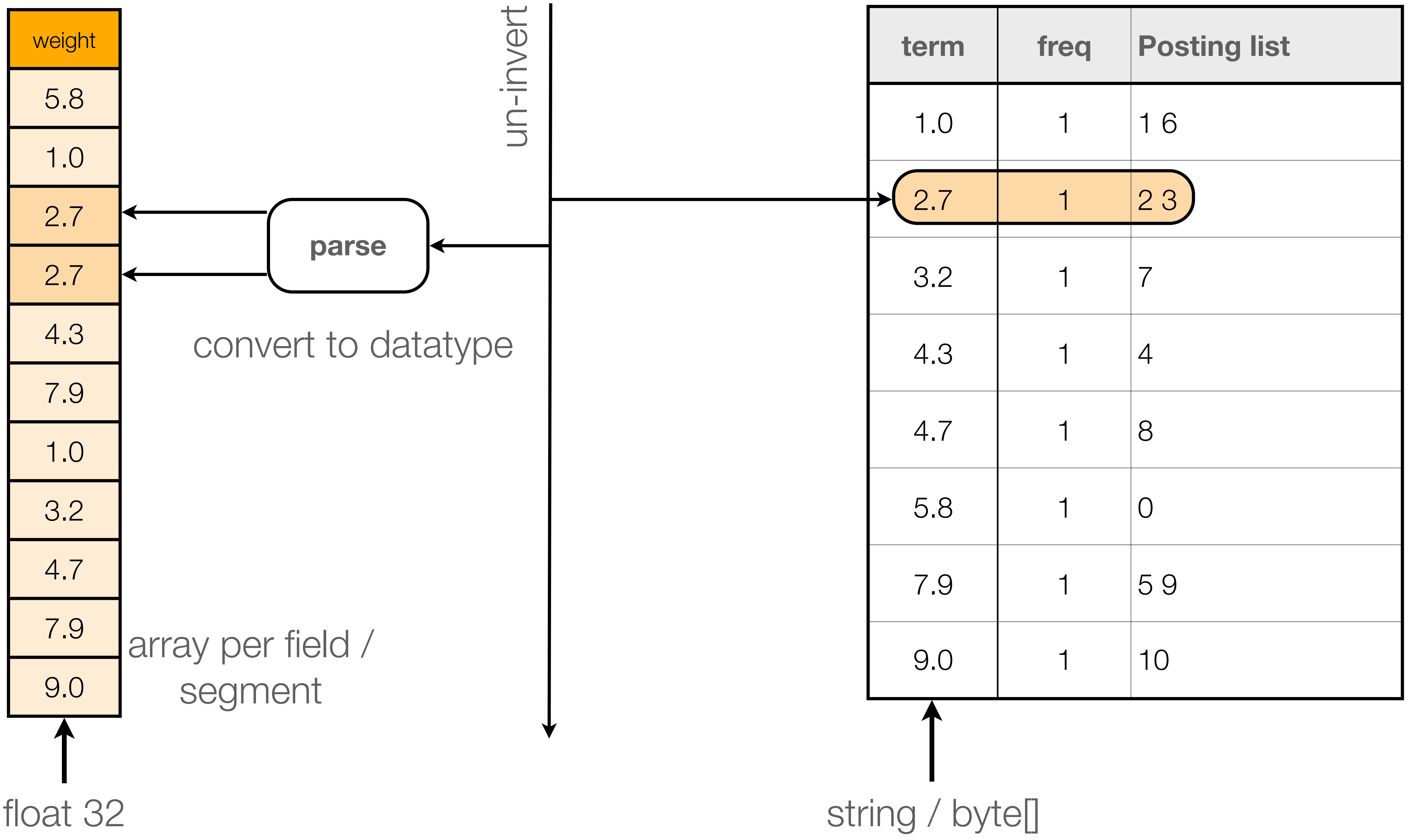
FieldCache

Awesome - lets undo all the indexing work!

Problem here: this works well :(

Uninverting a Field

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

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

FieldCache - loading

Simple Benchmark

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

100k Docs	1M Docs	10M Docs
122 ms	348 ms	3161 ms

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

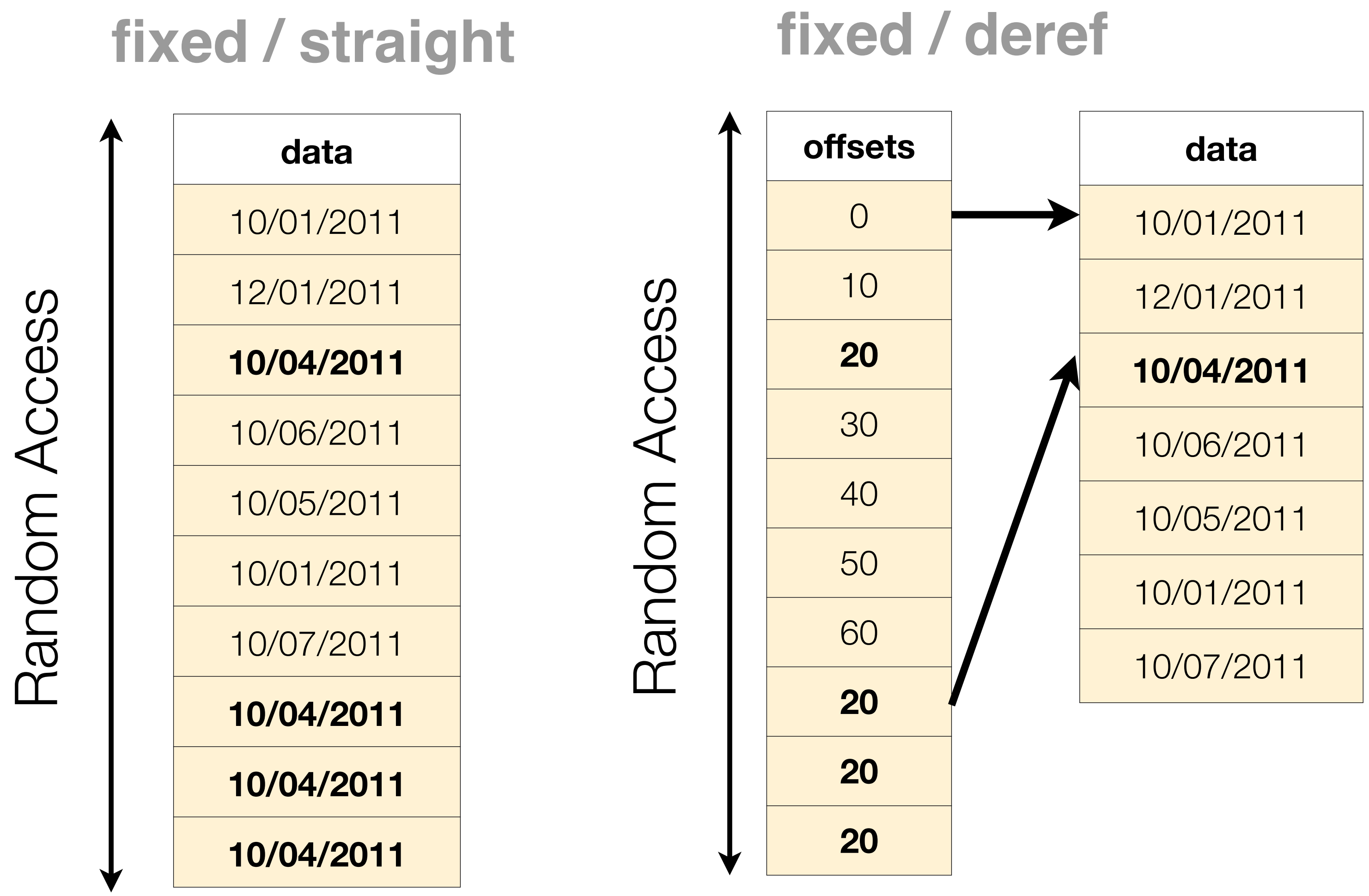
1 value per document

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

integer integer float 32

Arbitrary Values - The byte[] variants

- Length Variants:
  - **Fixed / Variable**
- Store Variants:
  - **Straight or Referenced**



fixed / straight

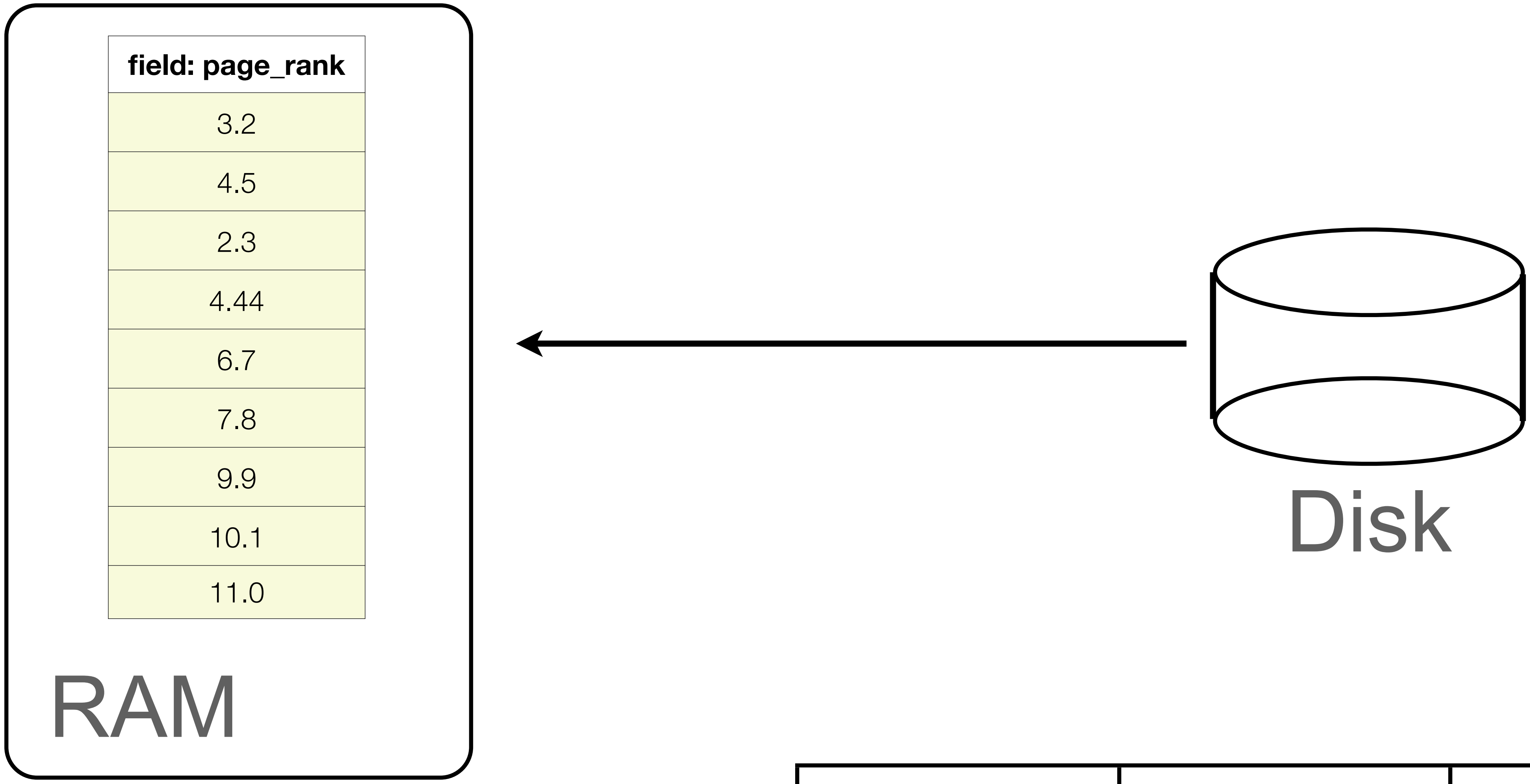
data
10/01/2011
12/01/2011
10/04/2011
10/06/2011
10/05/2011
10/01/2011
10/07/2011
10/04/2011
10/04/2011
10/04/2011

fixed / deref

offsets
0
10
20
30
40
50
60
20
20
20

data
10/01/2011
12/01/2011
10/04/2011
10/06/2011
10/05/2011
10/01/2011
10/07/2011

IndexDocValues - loading



field: page_rank
3.2
4.5
2.3
4.44
6.7
7.8
9.9
10.1
11.0

RAM

Disk

	100k Docs	1M Docs	10M Docs
FieldCache	122 ms	348 ms	3161 ms
DocValues	**7 ms**	**10 ms**	**90 ms**

Selective in-memory / on-disk Access

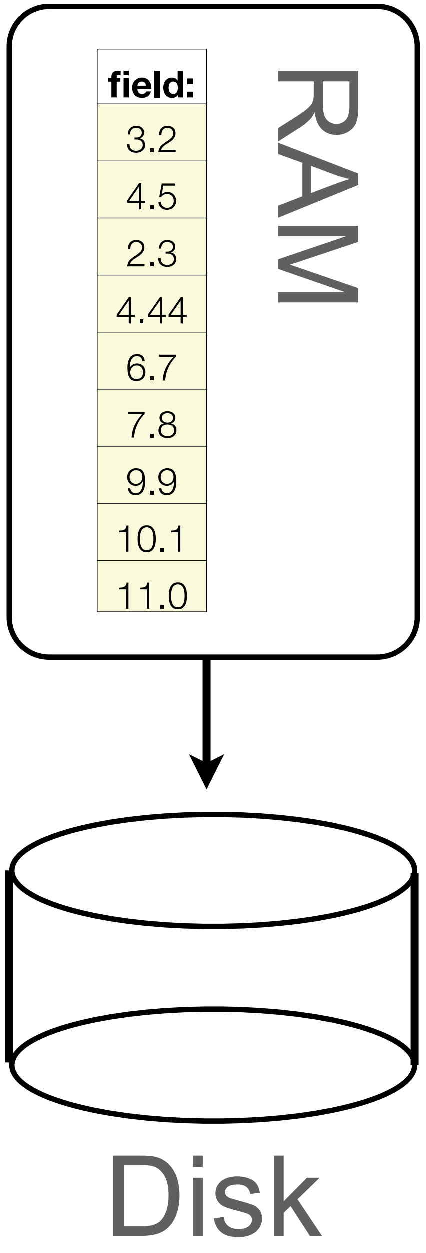
```
IndexReader reader;
IndexDocValues docValues = reader.docValues("page_rank");
Source source = docValues.getSource();
```

loads in RAM on first access

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

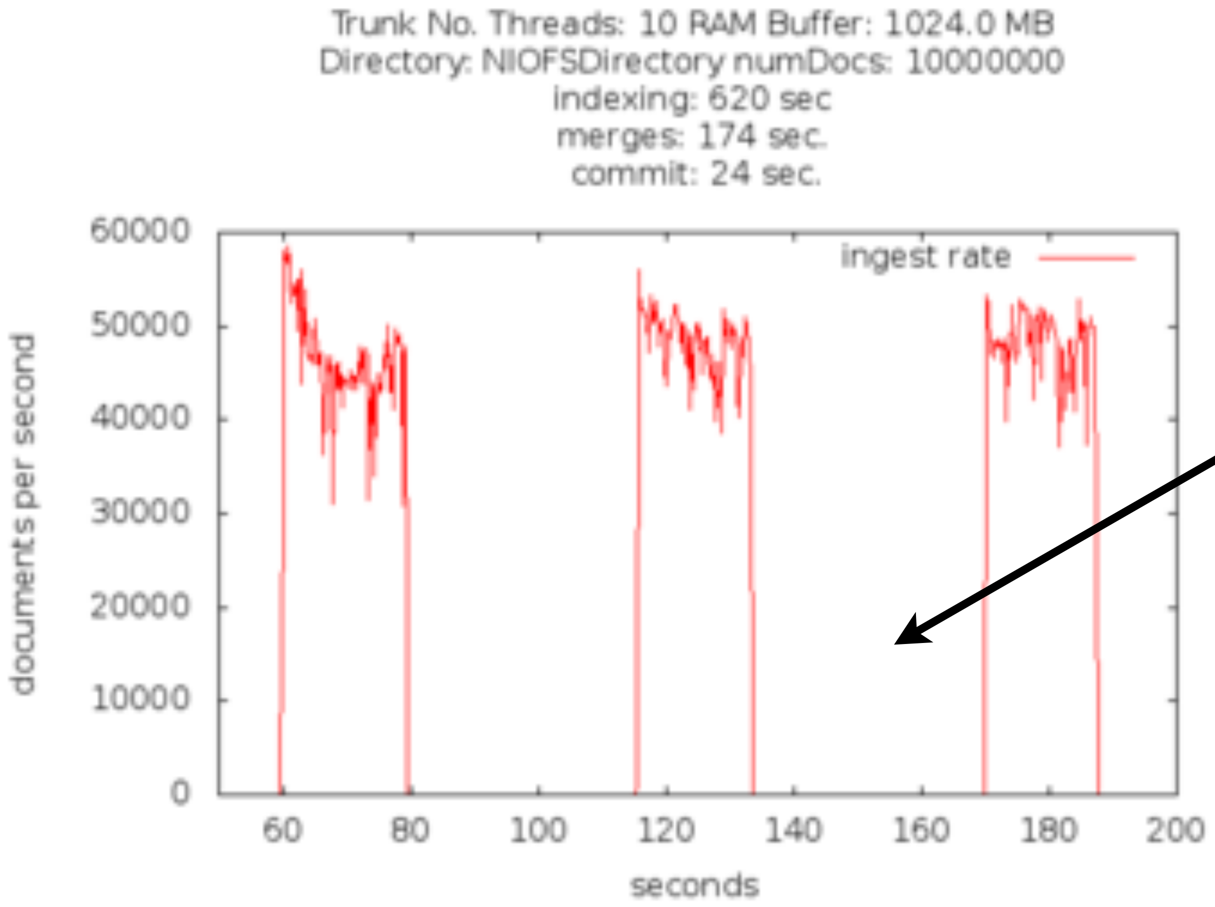
goes to disk directly

performance hit 40 - 80% (YMMV)



field:
3.2
4.5
2.3
4.44
6.7
7.8
9.9
10.1
11.0

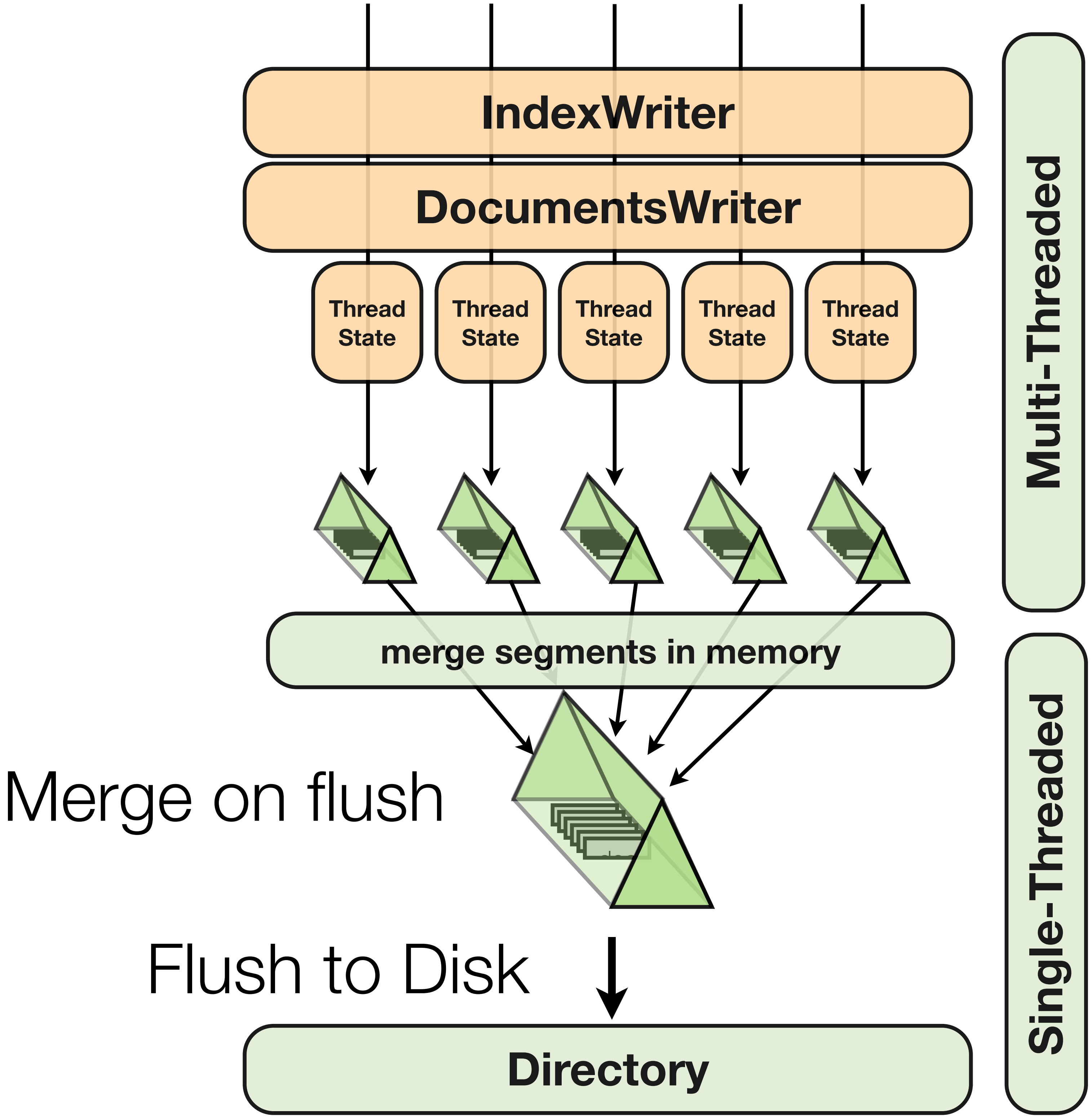
DocumentsWriterPerThread



Question: WTF is the IndexWriter doing there?

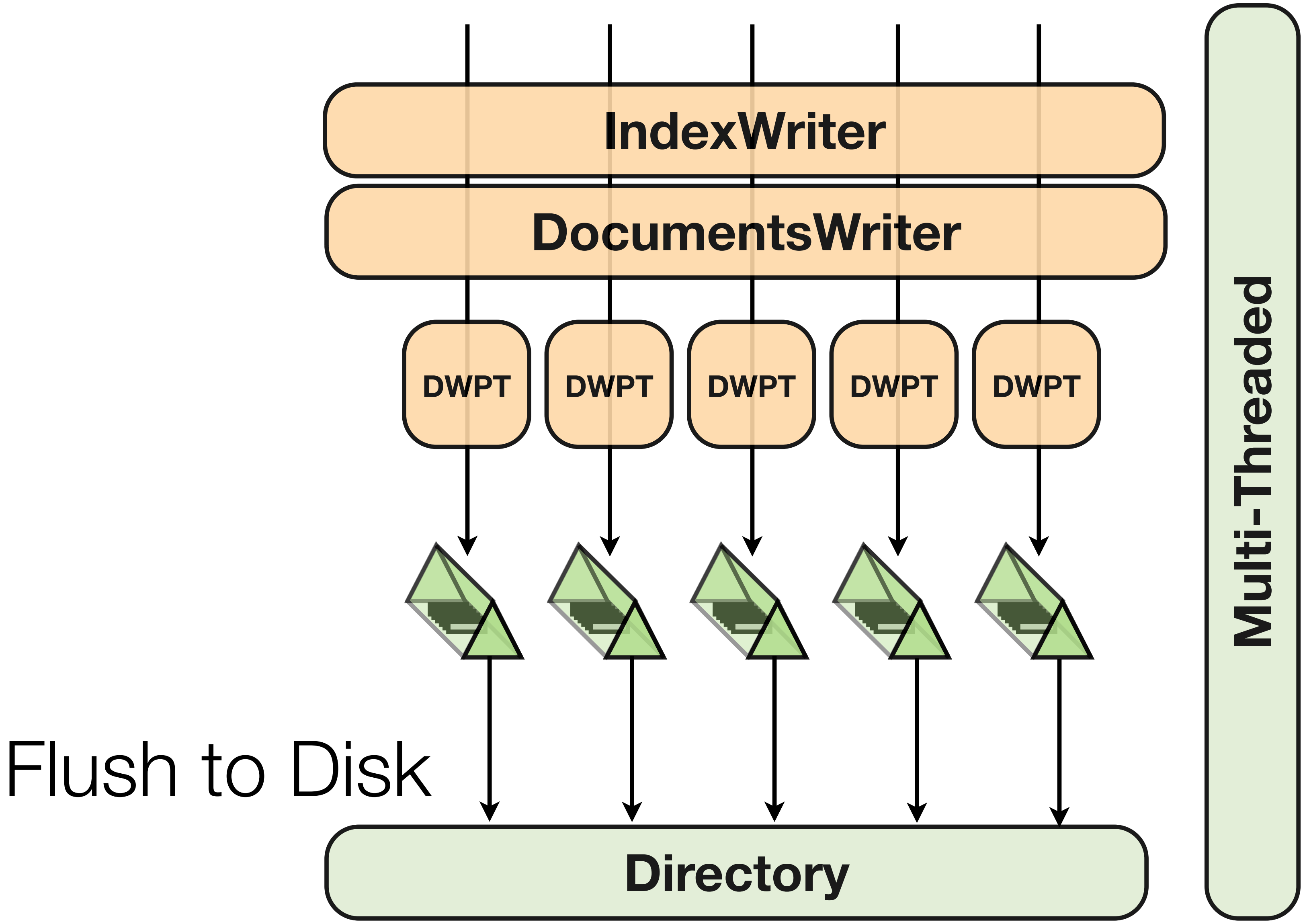
Indexing Ingest Rate over time with Lucene 3.x Indexing 7 Million 4kb wikipedia documents

A whole lot of nothing.... prior to DWPT

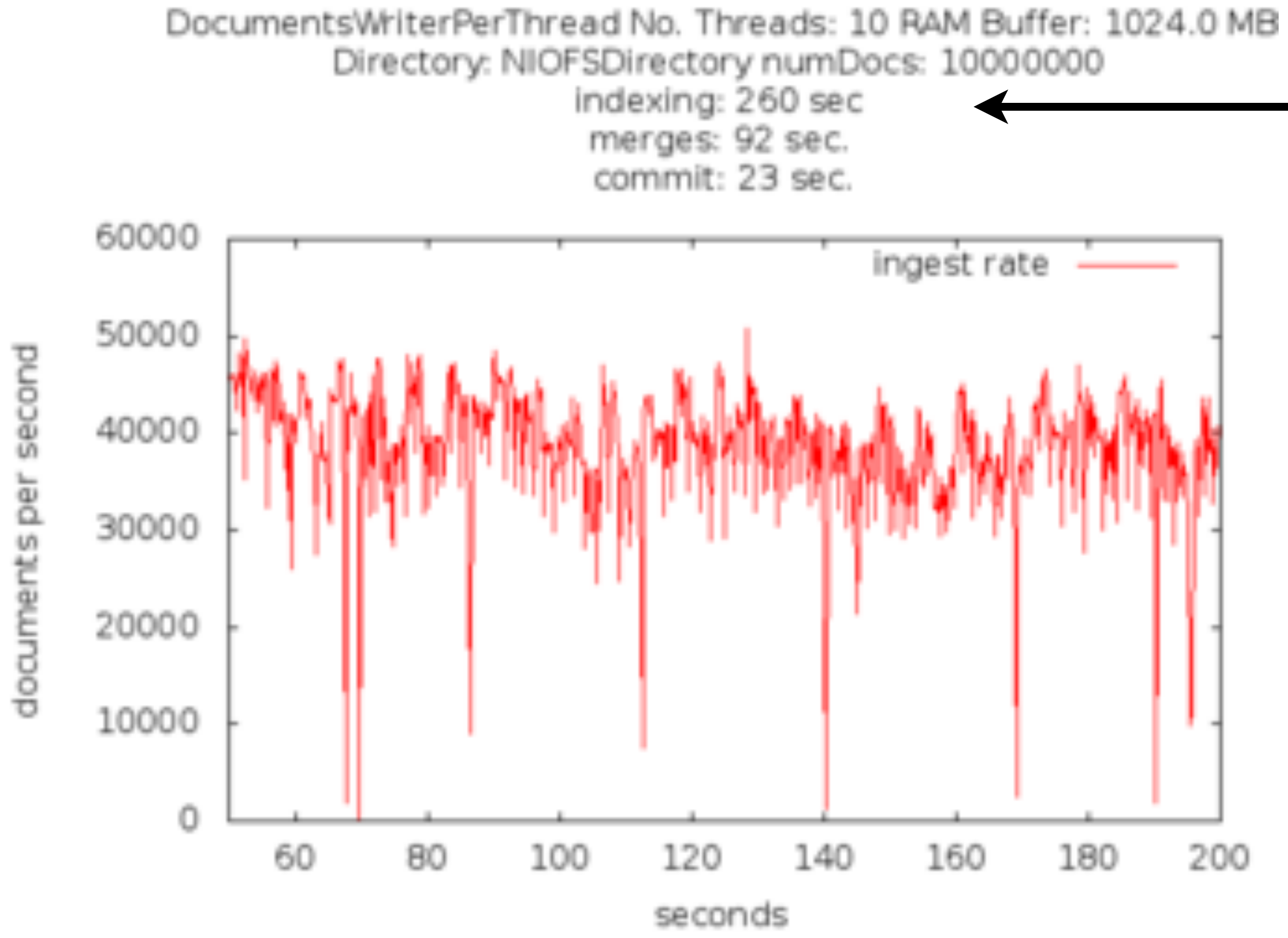


Answer: it gives you threads a break and it's having a drink with your slow-as-s**t IO System

Keep you resources busy with DWPT



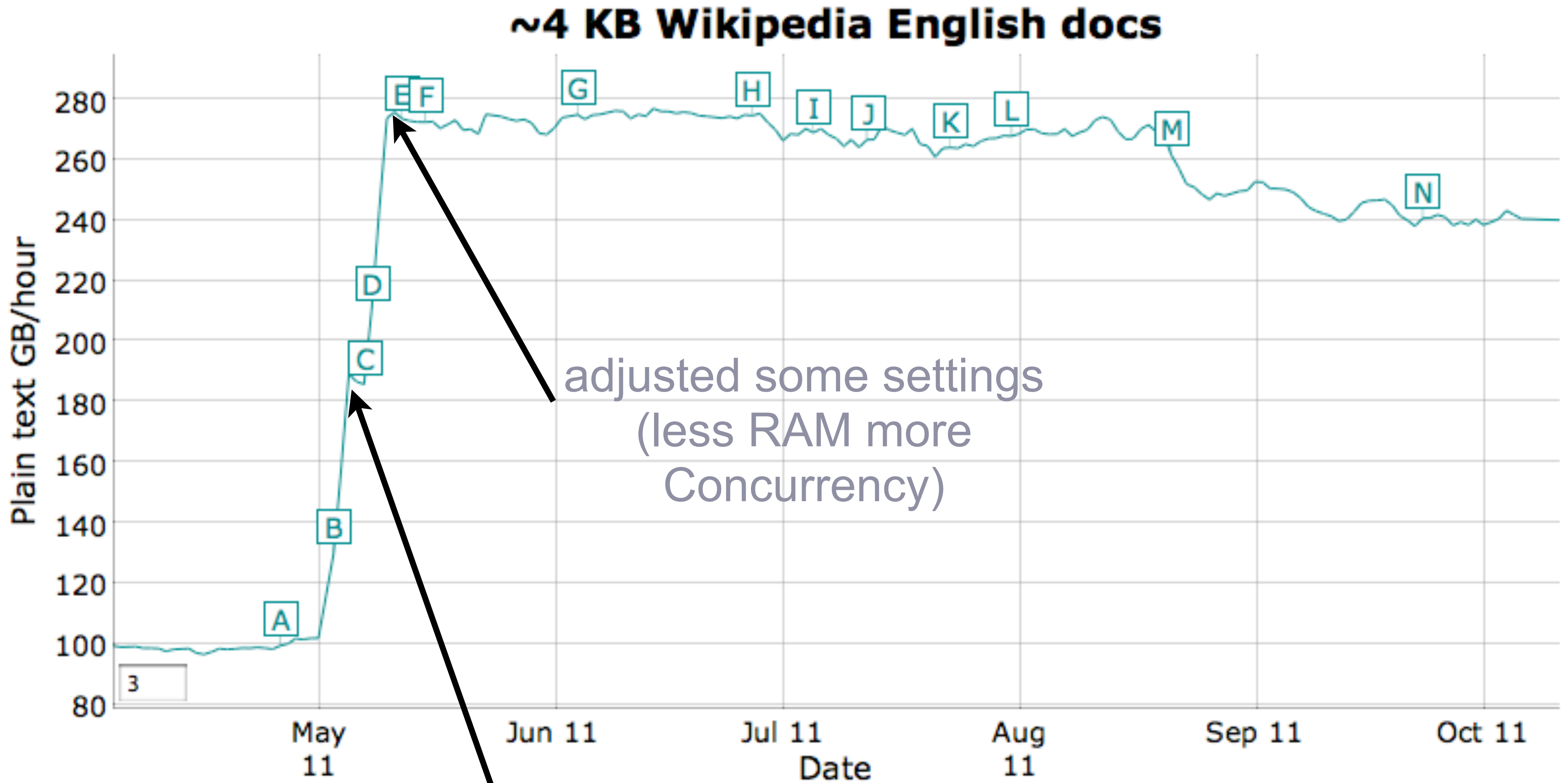
Title Text



vs. 620 sec on 3.x

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

280% improvement

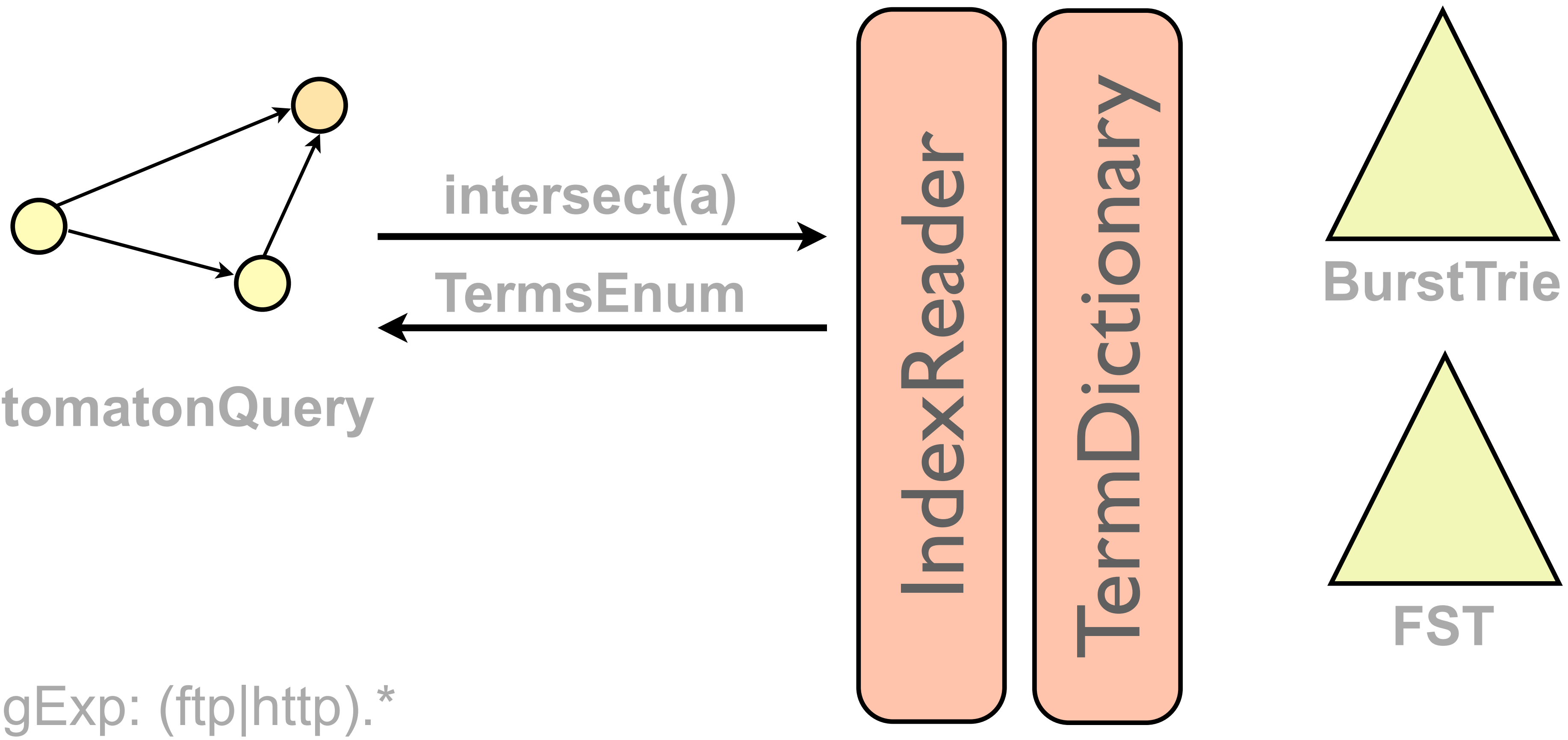


adjusted some settings (less RAM more Concurrency)

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



intersect(a)

TermsEnum

AutomatonQuery

IndexReader

TermDictionary

BurstTrie

FST

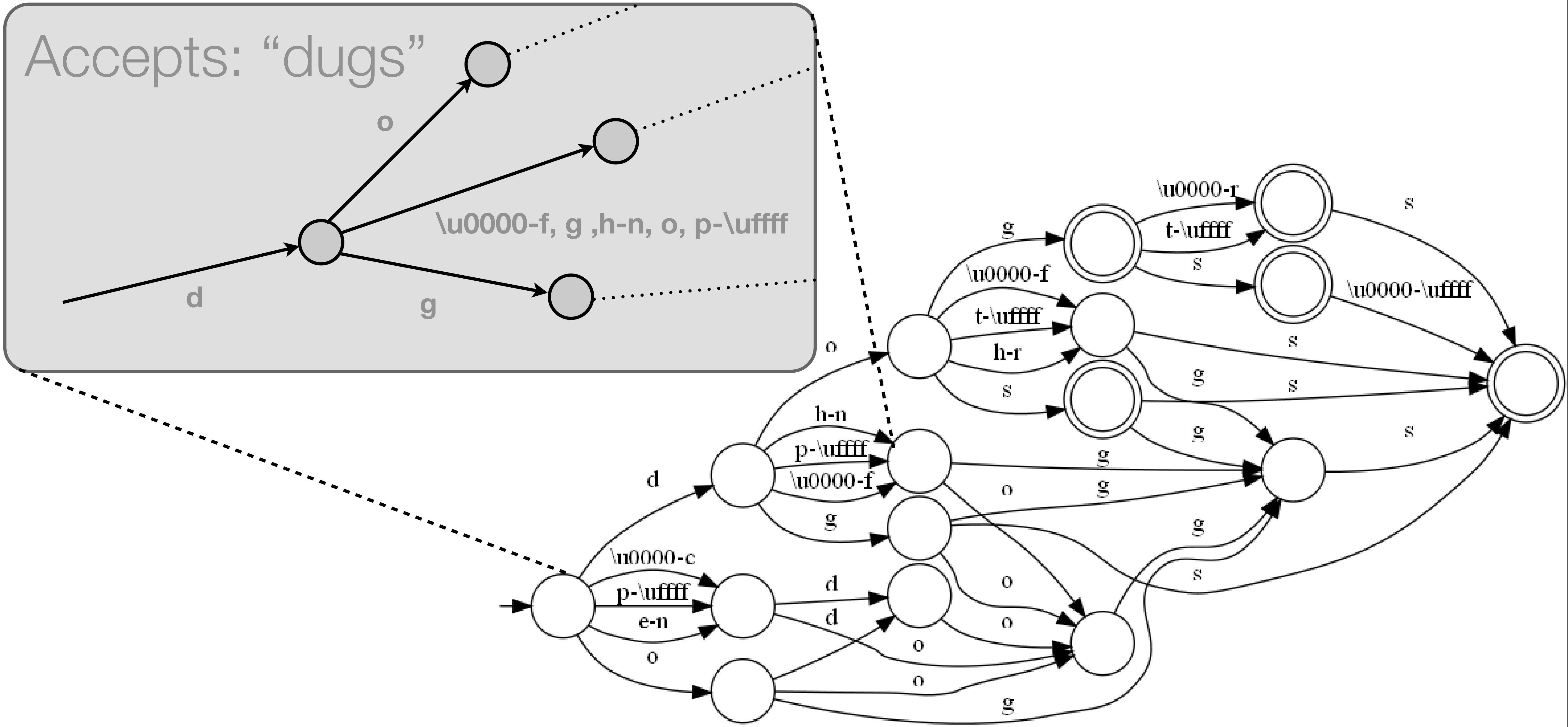
RegExp: (ftp|http).*

Fuzzy: dogs~1

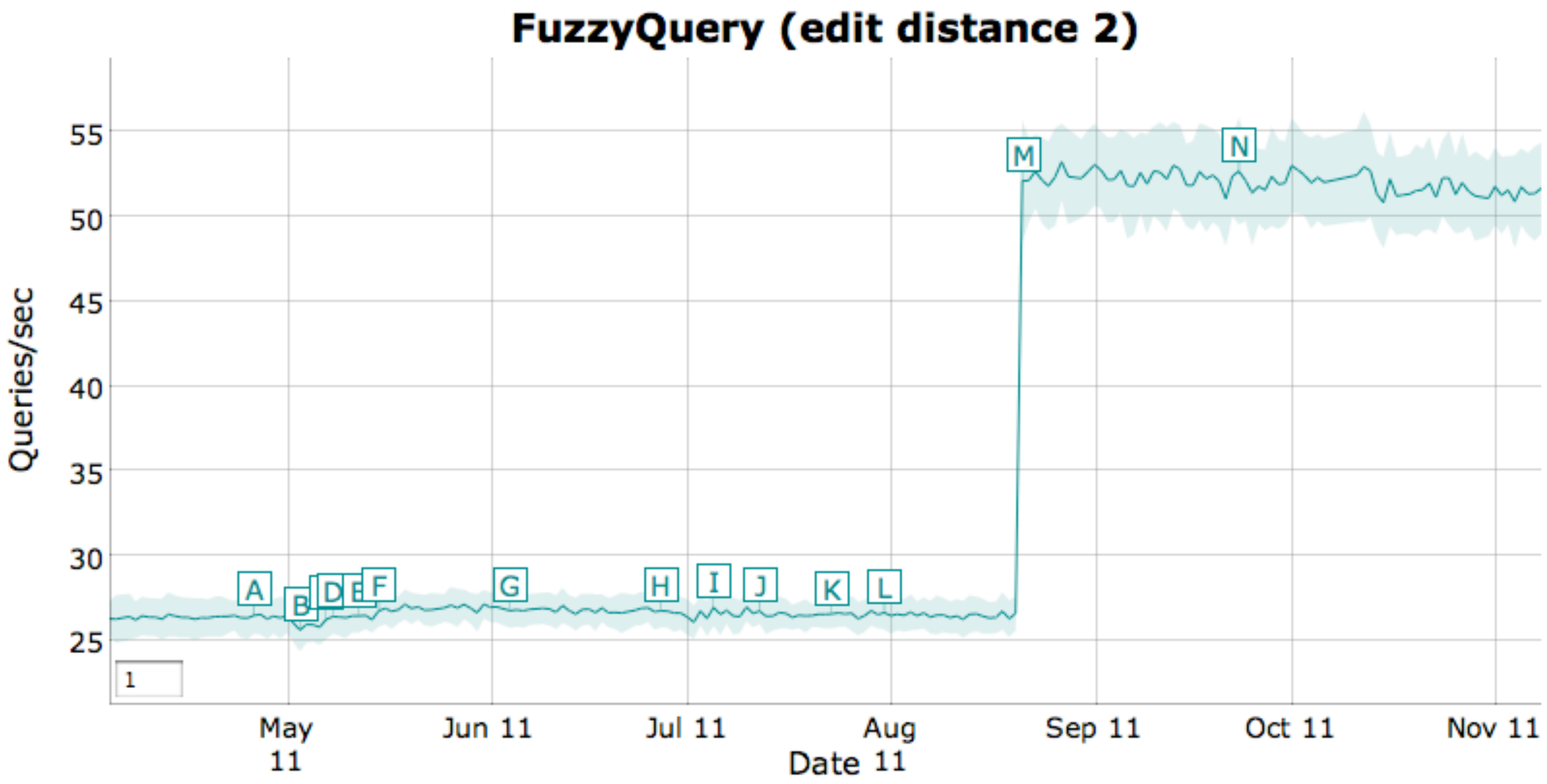
Fuzzy-Prefix: (dogs~1).*

Automaton Queries (Fuzzy)

Example DFA for “dogs” Levenshtein Distance 1



Here are the 20k % everybody waits for :D



In Lucene 3 this is about 0.1 - 0.2 QPS

Composing your own AutomatonQuery

```
// 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

Questions

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 every day code
- Prevents leaking file handles (random exception testing)
- Gained confidence that if there is a bug we gonna hit it one day