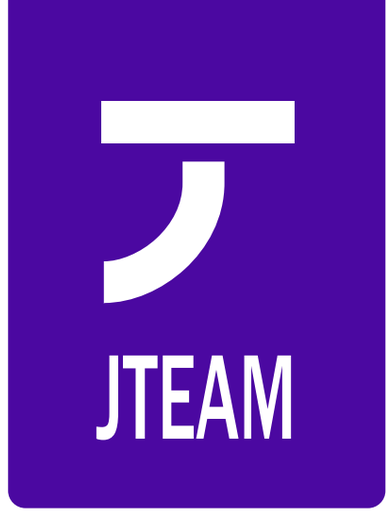


Column Stride Fields aka. DocValues

Simon Willnauer @ BerlinBuzzwords 2011

PMC Member & Core Comitter Apache Lucene
simon@apache.org / simon@jteam.nl

Column Stride Fields aka. DocValues

JTEAM

Agenda

- **What is this all about? aka. The Problem!**
- The more native solution
- DocValues - current state and future
- Questions?

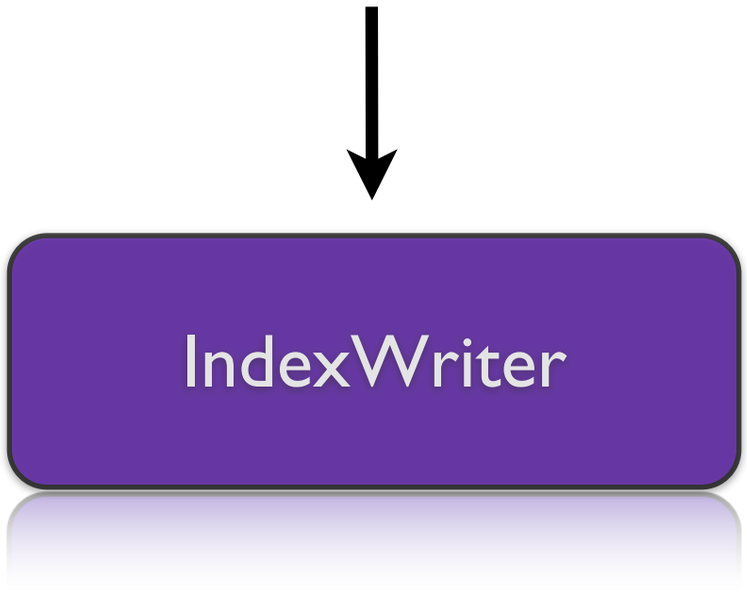
What is this all about? - Inverted Index

JTEAM

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

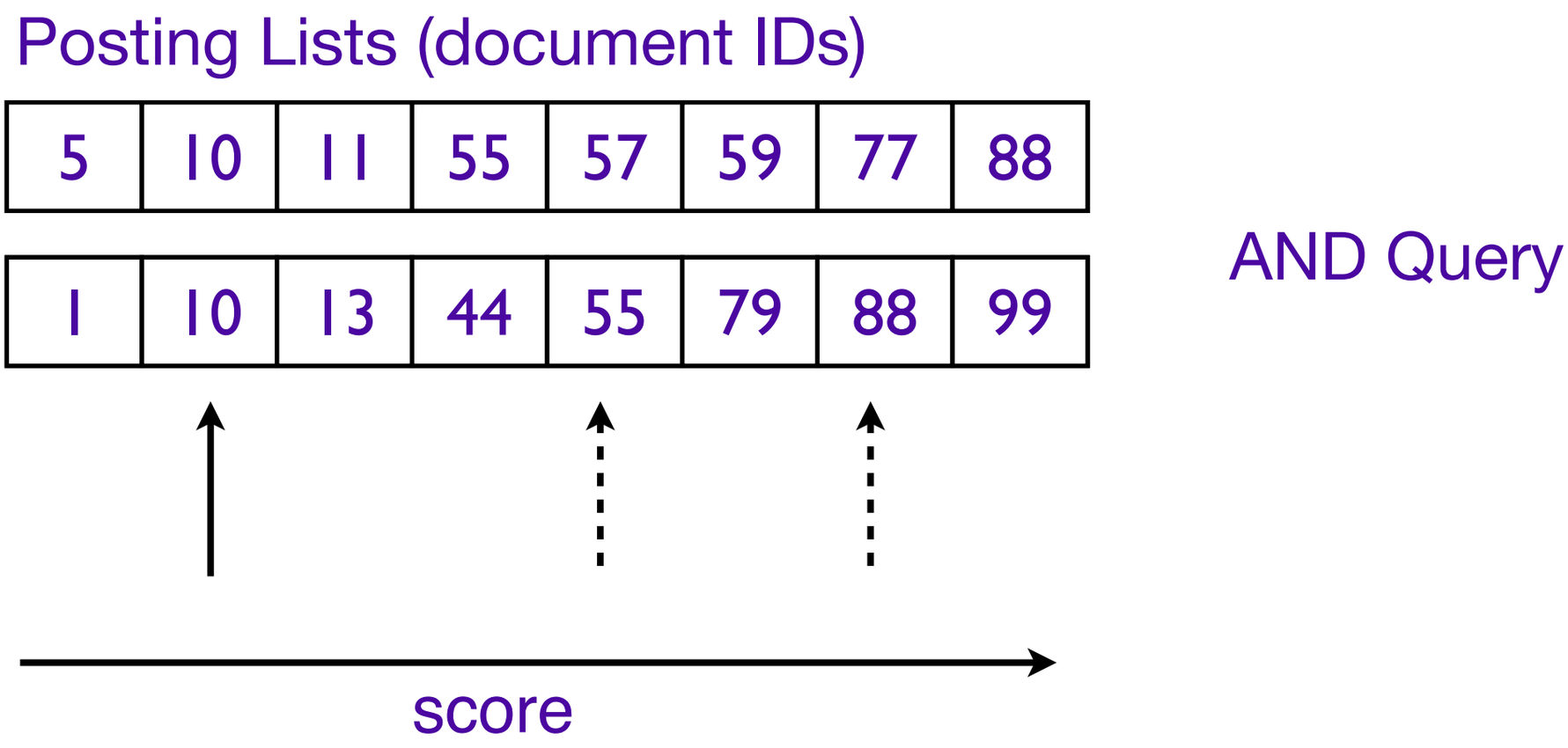


TermsEnum

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....



What goes into the score? **PageRank?**, **ClickFeedback?**

How to store scoring factors?

Lucene provides 2 ways of storing data

- **Stored Fields** (document to String or Binary mapping)
- **Inverted Index** (term to document mapping)

What we need here is one or more values per document!

- document to value mapping

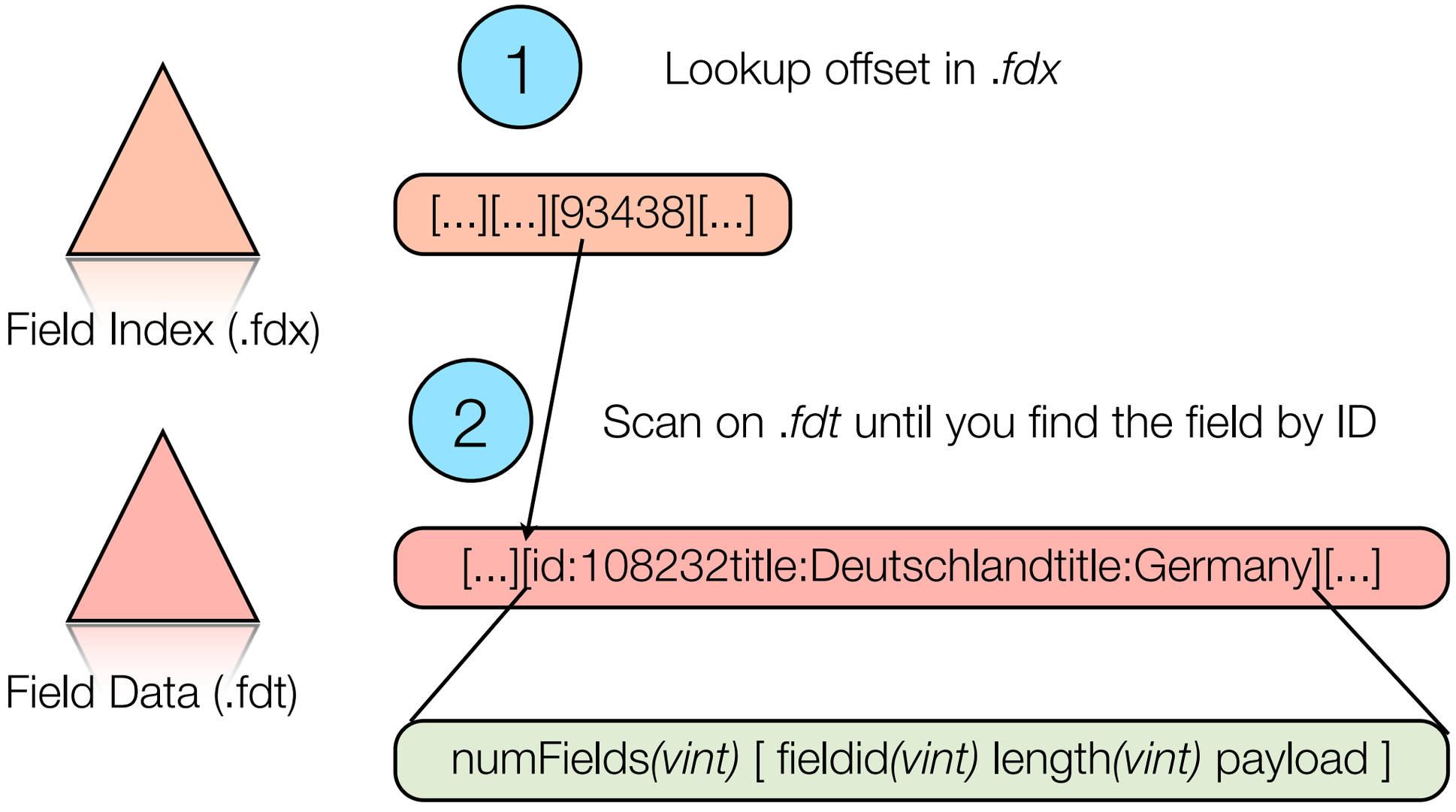
Why not use **Stored Fields**?

Using Stored Fields

JTEAM

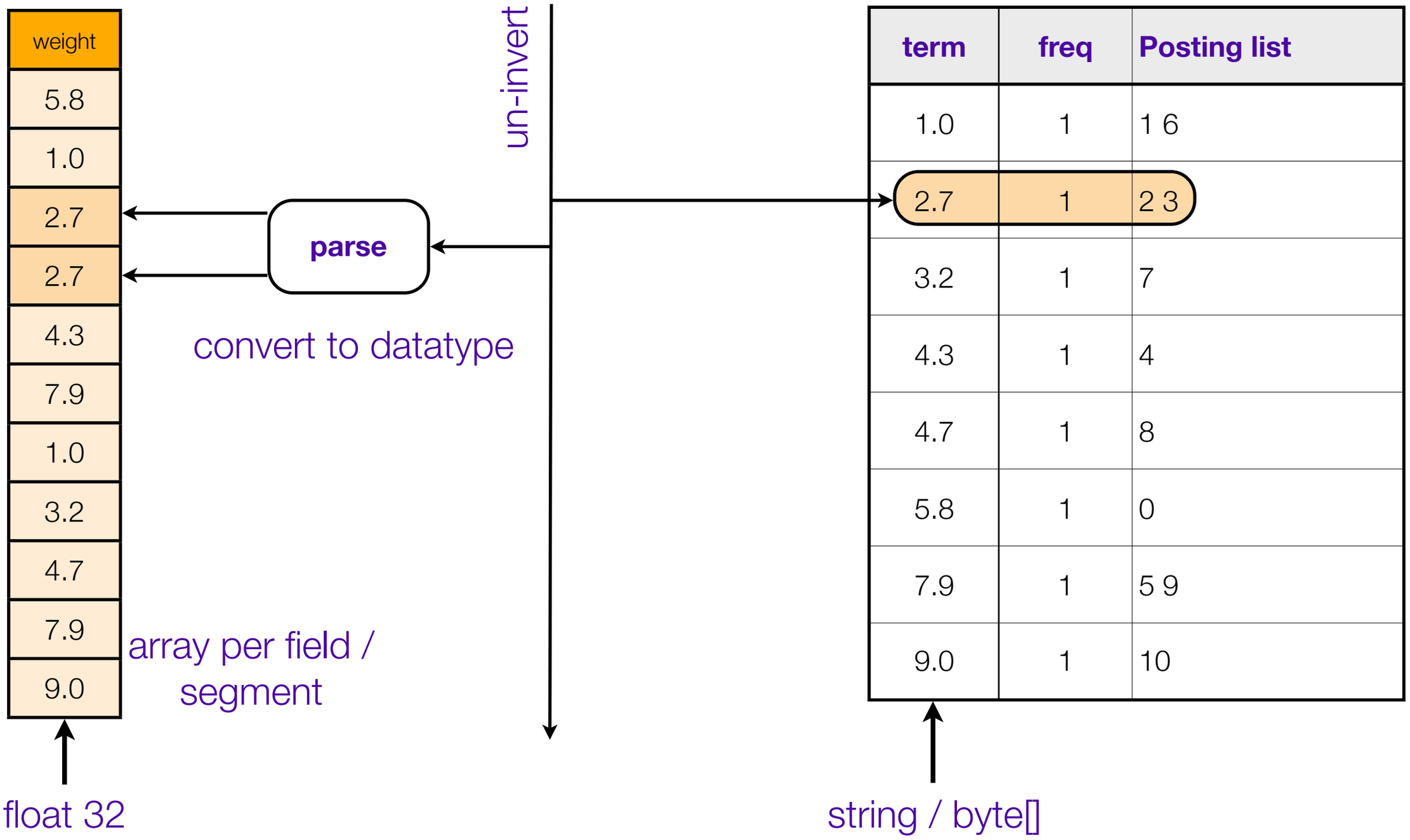
- **Stored Fields** serve a different purpose
  - loading *body* or *title* fields for result rendering / highlighting
  - very suited for loading multiple values
- With **Stored Fields** you have one indirection per document resulting in going to disk twice for each document
  - on-disk random access is too slow
  - remember **Lucene** could score millions of documents even if you just render the top 10 or 20!

Accessing a Stored Field Value



Alternatives?

Lucene can un-invert a field into **FieldCache**



FieldCache - is fast once loaded, once!

- Constant time lookup **DocID** to **value**
- Efficient representation
  - primitive array
  - low GC overhead
- **loading can be slow (realtime can be a problem)**
- **must parse values**
- **builds unnecessary term dictionary**
- **always memory resident**

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**

FieldCache works fine! - if...

- you have enough memory
- you can afford the loading time
- merge is fast enough (for FieldCache you need to index the terms)

What if you can't? Like when you are in a very restricted environment?

- 3 Billion Android installations world wide and growing - **2 MB Heap!**
- with 100 Million Documents one field takes **30 seconds** to load

Summary

- Stored Fields are not fast enough for random access
- FieldCache is fast once loaded
  - abuses a reverse index
  - must convert to String and from String
  - requires fair amount of memory
- Lucene is **missing** native data-structure for primitive per-document values

JTEAM

Agenda

- What is this all about? aka. The Problem!
- **The more native solution**
- DocValues - current state and future
- Questions?

The more native solution - Column Stride Fields

- A dense column based storage
- 1 value per document
- accepts primitives - no conversion from / to string
  - **int**
  - **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

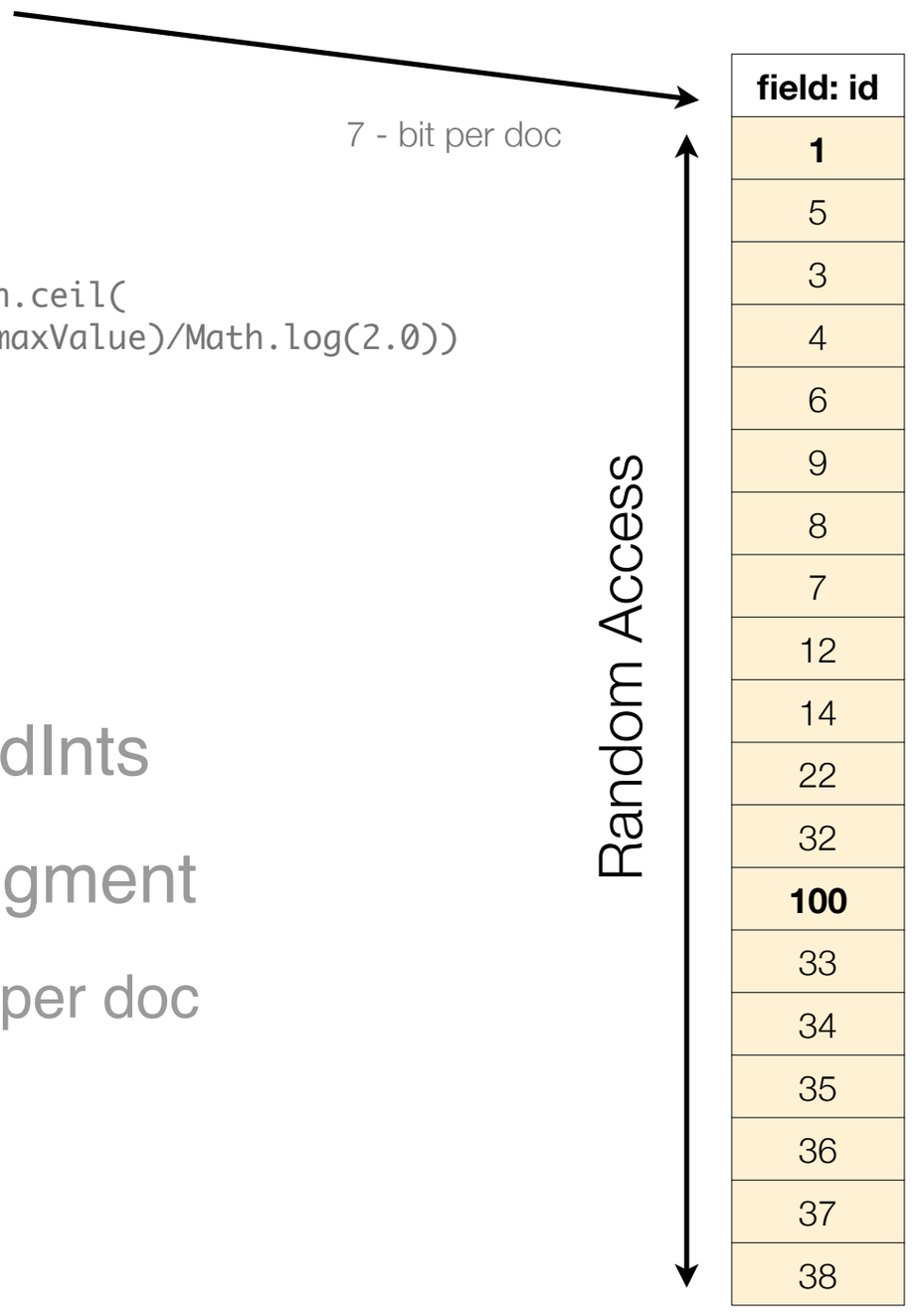
1 value per document

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**

Numeric Types - Int

Number of bit depend on the numeric range in the field:

```
Math.max(1, (int) Math.ceil(
          Math.log(1+maxValue)/Math.log(2.0))
          );
```



- **Integer** are stored dense based on PackedInts
- Space depends on the value-range per segment

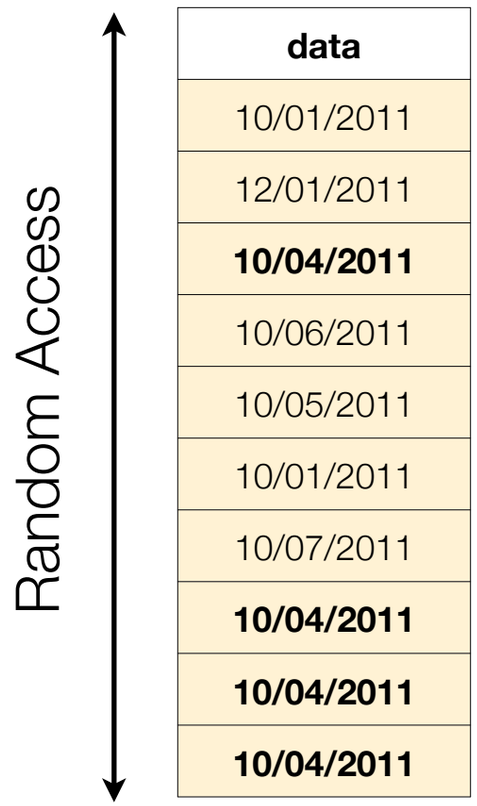
  Example: *[1, 100]* maps to *[0, 99]* requires 7 bit per doc

- **Floats** are stored without compression
  - either 32 or 64 bit per value

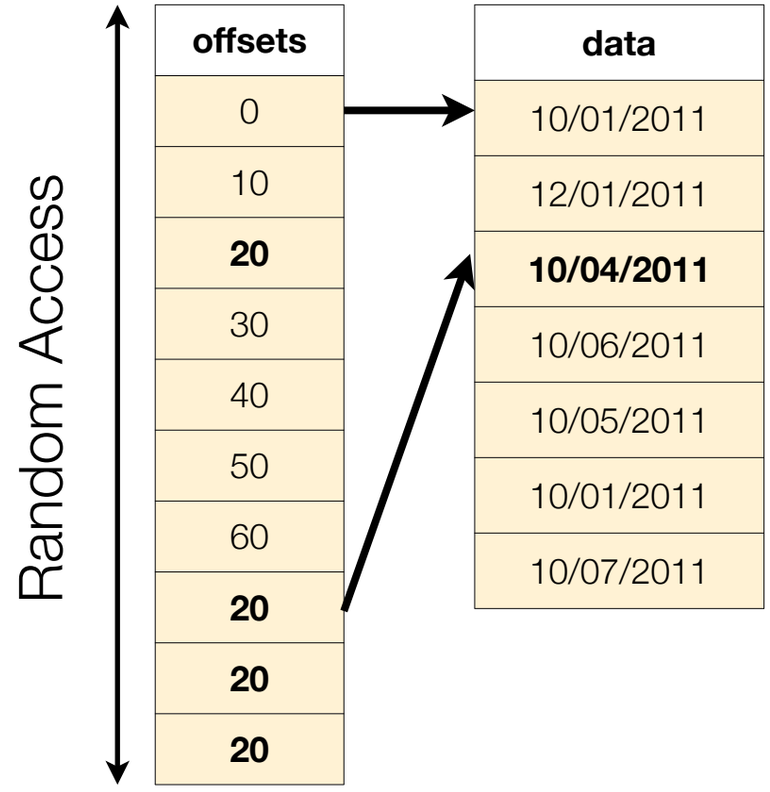
Arbitrary Values - The byte[] variants

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

fixed / straight



fixed / deref



IndexDocValues - Memory Requirements

- **RAM Resident - random access**
  - similar to FieldCache
  - bytes are stored in byte-block pools
  - currently limited to 2GB payload per segment
- **On-Disk - sequential access**
  - almost no JVM heap memory
  - files should be in FS cache for fast access

Lets look at the API - Indexing

Adding DocValues follows existing patterns, simply use **Fieldable**

```
Document doc = new Document();
float pageRank = 10.3f;
IndexDocValuesField valuesField = new IndexDocValuesField("pageRank");
valuesField.setFloat(pageRank);
doc.add(valuesField);
writer.addDocument(doc);
```

Sometimes the field should also be **indexed**, **stored** or needs **term-vectors**

```
String titleText = "The quick brown fox";
Field field = new Field("title", titleText , Store.NO, Index.ANALYZED);
IndexDocValuesField titleDV = new IndexDocValuesField("title");
titleDV.setBytes(new BytesRef(titleText), Type.BYTES_VAR_DEREF);
field.setDocValues(titleDV);
```

Looking at the API - Search / Retrieve

JTEAM

On disk sequential access is exposed through **ValuesEnum**

```
IndexReader reader = ...;
IndexDocValues values = reader.docValues("pageRank");
ValuesEnum floatEnum = values.getEnum();
int doc = 0;
FloatsRef ref = floatEnum.getFloat(); // values are filled when iterating

while((doc = floatEnum.nextDoc()) != ValuesEnum.NO_MORE_DOCS) {
  double value = ref.floats[0];
}

// equivalent to ...

int doc = 0;
while((doc = floatEnum.advance(doc+1)) != ValuesEnum.NO_MORE_DOCS) {
  double value = ref.floats[0];
}
```

ValuesEnum is based on **DocIdSetIterator** just like **Scorer** or **DocsEnum**

Looking at the API - Search / Retrieve

JTEAM

RAM Resident API is very similar to **FieldCache**

```
IndexReader segmentReader = ...;
IndexDocValues values = segmentReader.perDocValues().docValues("pageRank");
Source source = values.getSource();
double value = source.getFloat(x);

// still allows iterating over the RAM resident values
```

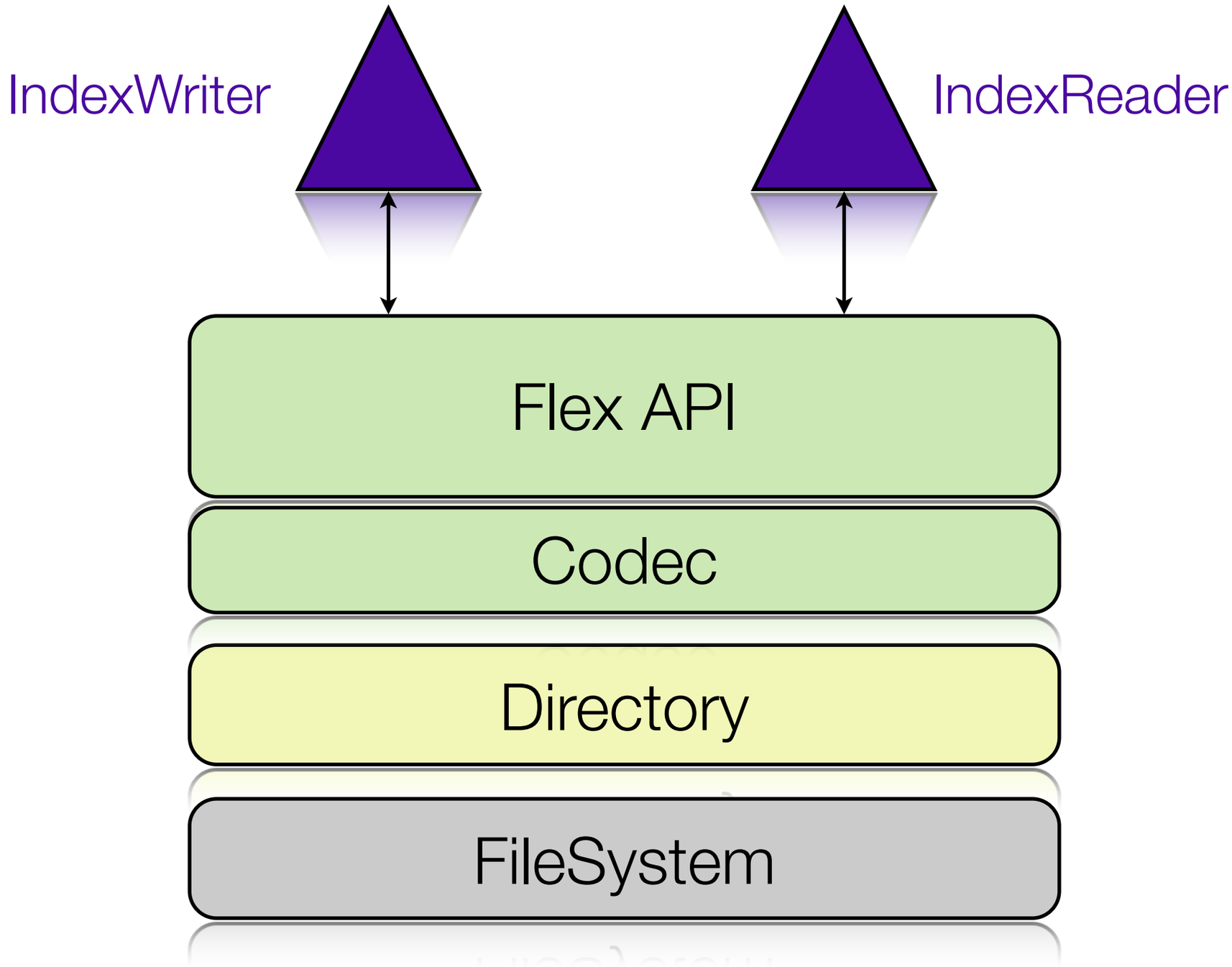
```
ValuesEnum floatEnum = source.getEnum();
int doc;
FloatsRef ref = floatEnum.getFloat();
while((doc = floatEnum.nextDoc()) != ValuesEnum.NO_MORE_DOCS) {
  value = ref.floats[0];
}
```

ValuesEnum still available on RAM Resident API

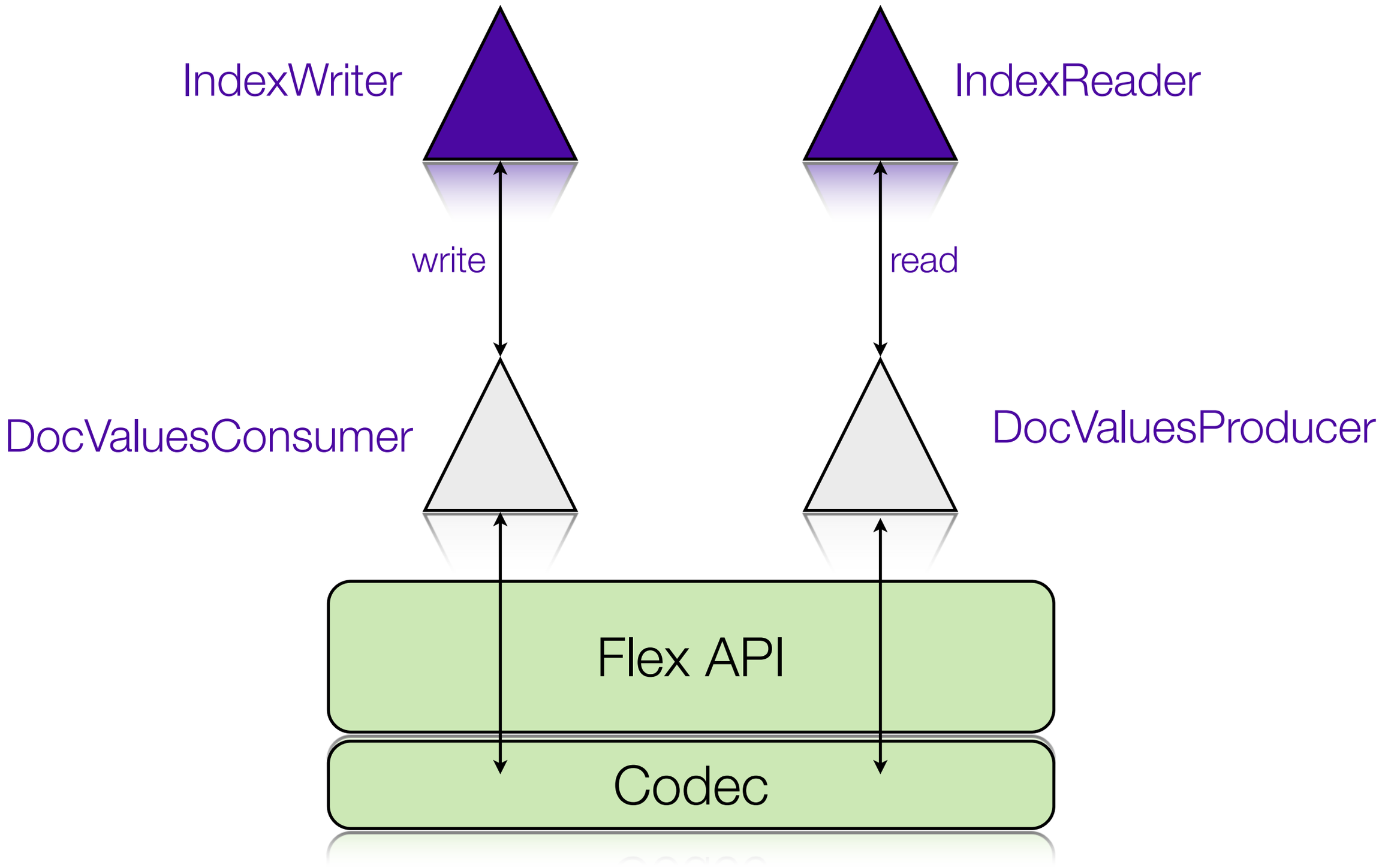
Can I add my own DocValues Implementation?

- DocValues are integrated into **Flexible Indexing**
- **IndexWriter / IndexReader** write and read DocValues via a **Codec**
- DocValues **Types** are fixed (int, float32, float64 etc.) but implementations are **Codec** specific
- A Codec provides access to **DocValuesComsumer** and **DocValuesProducer**
  - allows implementing application specific serialization
  - customize compression techniques

Quick detour - Codecs



Quick detour - Codecs



Remember the loading FieldCache benchmark?

JTEAM

Simple Benchmark

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

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

Loading is **100 x** faster - no un-inverting, no string parsing

Query impact FieldCache vs. DocValues

Task	QPS DocValues	QPS FieldCache	% change
AndHighHigh	3.51	3.41	2.9%
PKLookup	46.06	44.87	2.7%
AndHighMed	37.09	36.48	1.7%
Fuzzy2	17.70	17.50	1.1%
Fuzzy1	27.15	27.21	-0.2%
Phrase	4.12	4.13	-0.2%
SpanNear	2.00	2.01	-0.5%
SloppyPhrase	1.98	2.02	-2.0%
Term	35.29	36.05	-2.1%
OrHighMed	4.73	4.93	-4.1%
OrHighHigh	3.99	4.18	-4.5%
Wildcard	12.97	13.60	-4.6%
Prefix3	15.86	16.70	-5.0%
IntNRQ	2.72	2.91	-6.5%

6 Search Threads 20 JVM instances, 5 instances per task run 50 times on 12 core Xeon / 24 GB RAM - all queries wrapped with a **CustomScoreQuery**

JTEAM

Agenda

- What is this all about? aka. The Problem!
- The more native solution
- **DocValues - current state and future**
- Questions?

DocValues - current state

- **Currently still in a branch**
  - Some minor JavaDoc issues
  - needs some cleanups
- **Landing on trunk very soon**
  - issue is already opened and active

DocValues - current features

- Fully customizable via Codecs
- User can control memory usage per field
- Suitable for environments where memory is tight
- Compact and native representation on disk and in RAM
- Fast Loading times
- Comparable to **FieldCache** (small overhead)

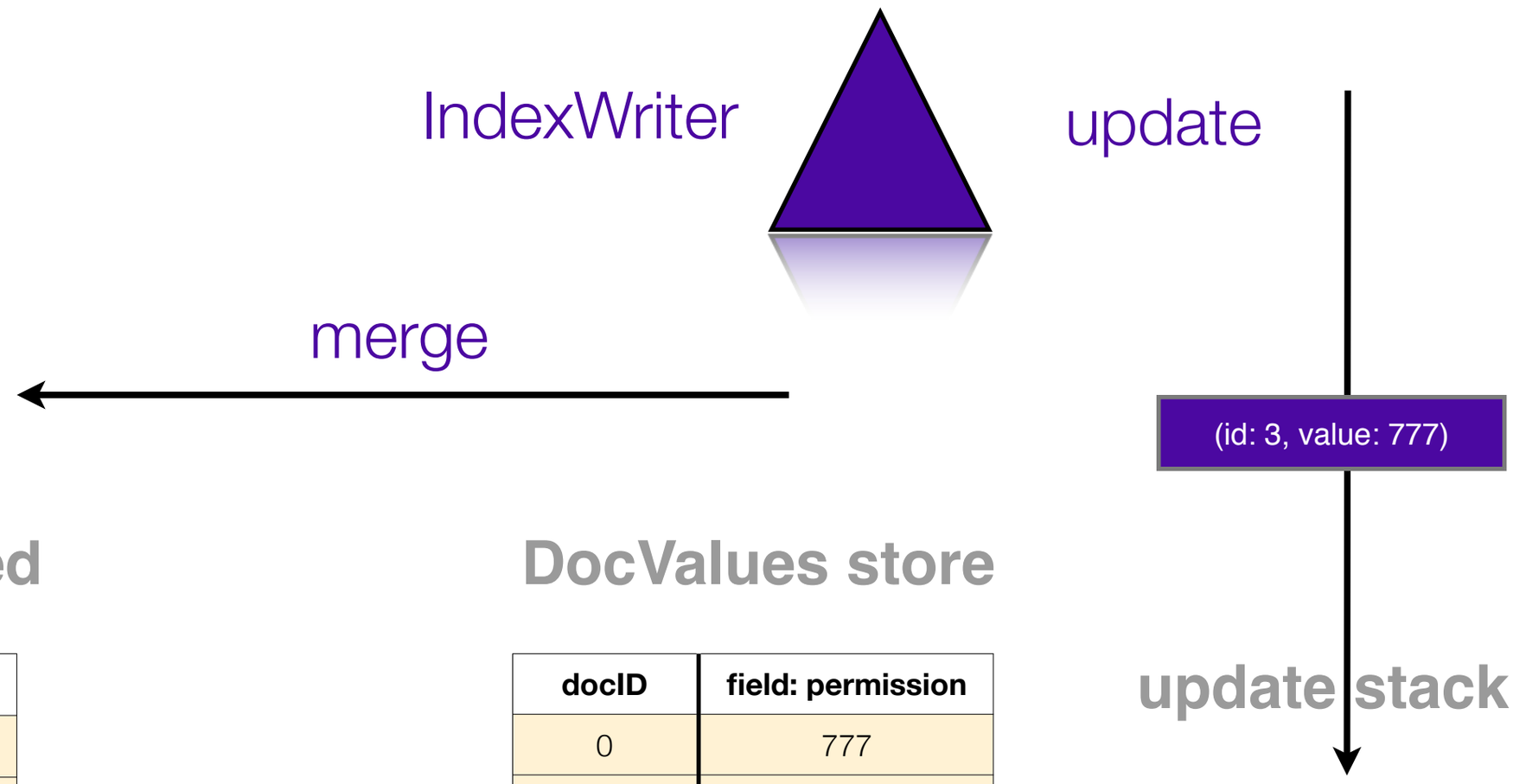
DocValues - what is next?

- until here there is no **huge** new value, right?
- the ultimate goal for **DocValues** is to be update-able
  - changing a per-document values without reindexing
  - users can replace existing values directly for each document
    - each field by itself will be update-able
- Will be available in **Lucene 4.0** once released ;)

DocValues - Updates

- **Lucene** has **write-once** policy for files
  - Changing in place is not a good idea - Consistency!
- Problem is comparable to **norms** or **deleted docs**
  - updating **norms** requires re-writing the entire **norms** array (1 byte per Document)
  - same is true for deleted docs while cost is low (1 bit per document)
- DocValues will use a **stacked-approach** instead

DocValues - Updates



coalesced stored

docID	field: permission
0	777
1	707
2	644
3	**777**
4	777
5	**644**
6	**777**
...	
n	

DocValues store

docID	field: permission
0	777
1	707
2	644
3	644
4	777
5	664
6	664

Use-Cases

- Scoring based on frequently changing values
  - click feedback
  - iterative algorithms like page rank
  - user ratings
- Restricted environments like Android
- Realtime Search (fast loading times)
- frequently changing fields
  - if the fields content is not searched!

269% Indexing Throughput speedup

Make sure you attend the Lightning-Talks at 4:20 PM today!

Questions?

Thank you for your attention!