Updateable fields in Lucene

and other Codec applications

Andrzej Białecki

ab@lucidimagination.com

Agenda

- Codec API primer
- Some interesting Codec applications
  - TeeCodec and TeeDirectory
  - FilteringCodec
  - Single-pass IndexSplitter
- Field-level updates in Lucene
  - Current document-level update design
  - Proposed “stacked” design
  - Implementation details and status
  - Limitations

About the speaker

- Lucene user since 2003 (1.2-dev…)
- Created Luke – the Lucene Index Toolbox
- Apache Nutch, Hadoop, Solr committer, Lucene PMC member, ASF member
- LucidWorks developer

Codec API

Data encoding and file formats

- Lucene 3.x and before
  - **Tuned to pre-defined data types**
  - **Combinations of delta encoding and variable-length byte encodings**
  - **Hardcoded choices – impossible to customize**
  - **Dependencies on specific file-system behaviors (e.g. seek back & overwrite)**
  - **Data coding happened in many places**
- Lucene 4 and onwards
  - **All data writing and reading abstracted from data encoding (file formats)**
  - **Highly customizable, easy to use API**

Codec API

- `Codec` implementations provide "formats"
  - **SegmentInfoFormat, PostingsFormat, StoredFieldsFormat, TermVectorFormat, DocValuesFormat**
- Formats provide consumers (to write to) and producers (to read from)
  - **FieldsConsumer, TermsConsumer, PostingsConsumer, StoredFieldsWriter / StoredFieldsReader …**
- Consumers and producers offer item-level API (e.g. to read terms, postings, stored fields, etc)

Codec Coding Craziness!

- Many new data encoding schemas have been implemented
  - **Lucene40, Pulsing, Appending**
- Still many more on the way!
  - **PForDelta, intblock Simple 9/16, VSEncoding, Bloom-Filter-ed, etc …**
- Lucene became an excellent platform for IR research and experimentation
  - **Easy to implement your own index format**

Some interesting Codec applications

TeeCodec

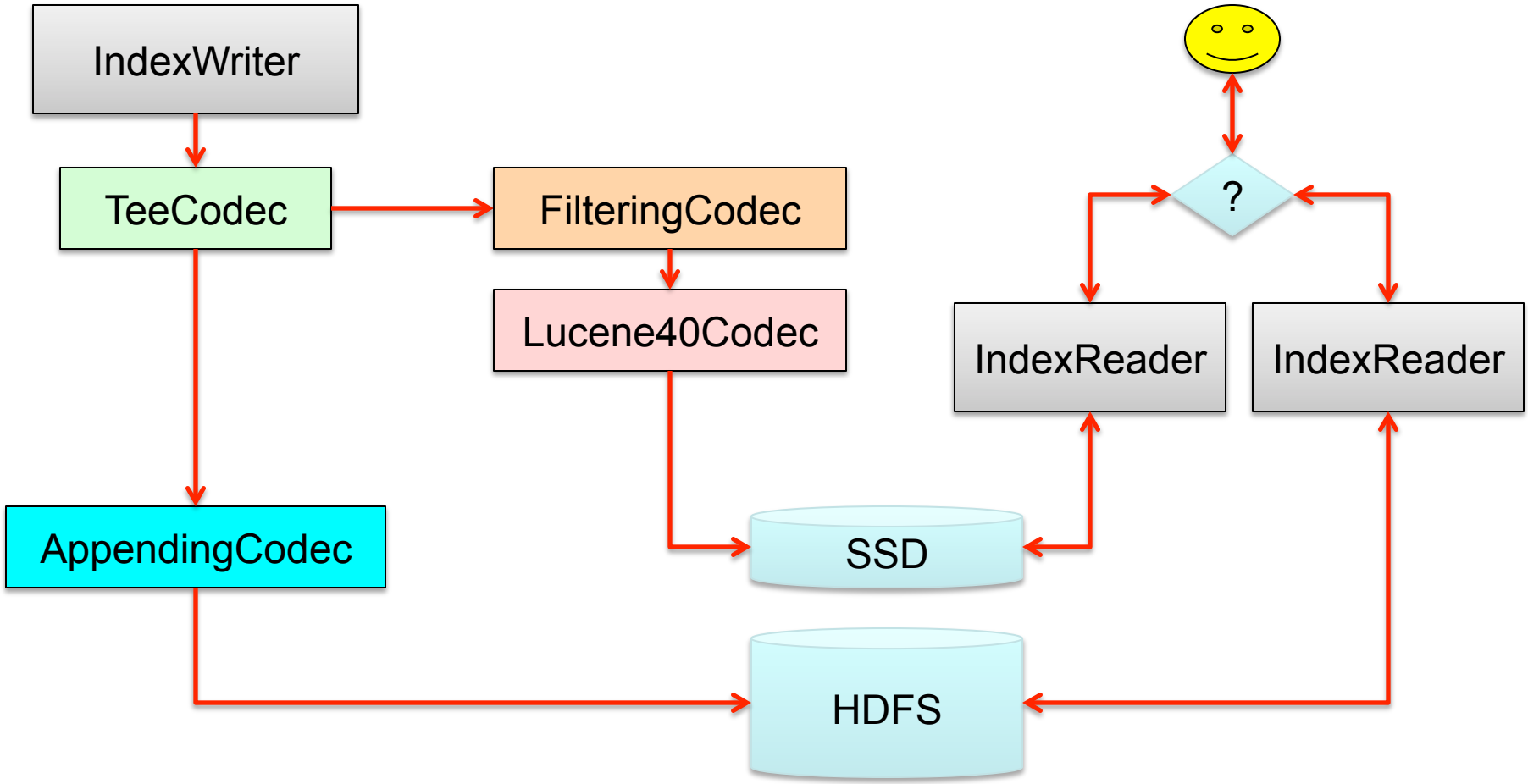
- Use cases:
  - **Copy of index in real-time, with different data encoding / compression**
- TeeCodec writes the same index data to many locations simultaneously
  - **Map<Directory,Codec> outputs**
  - **The same fields / terms / postings written to multiple outputs, using possibly different Codec-s**
- TeeDirectory replicates the stuff not covered in Codec API (e.g. segments.gen)

FilteringCodec

- Use case:
  - **Discard on-the-fly some less useful index data**
- Simple boolean decisions to pass / skip:
  - **Stored Fields (add / skip / modify fields content)**
  - **Indexed Fields (all data related to a field, i.e. terms + postings)**
  - **Terms (all postings for a term)**
  - **Postings (some postings for a terms)**
  - **Payloads (add / skip / modify payloads for term's postings)**
- Output: Directory + Codec

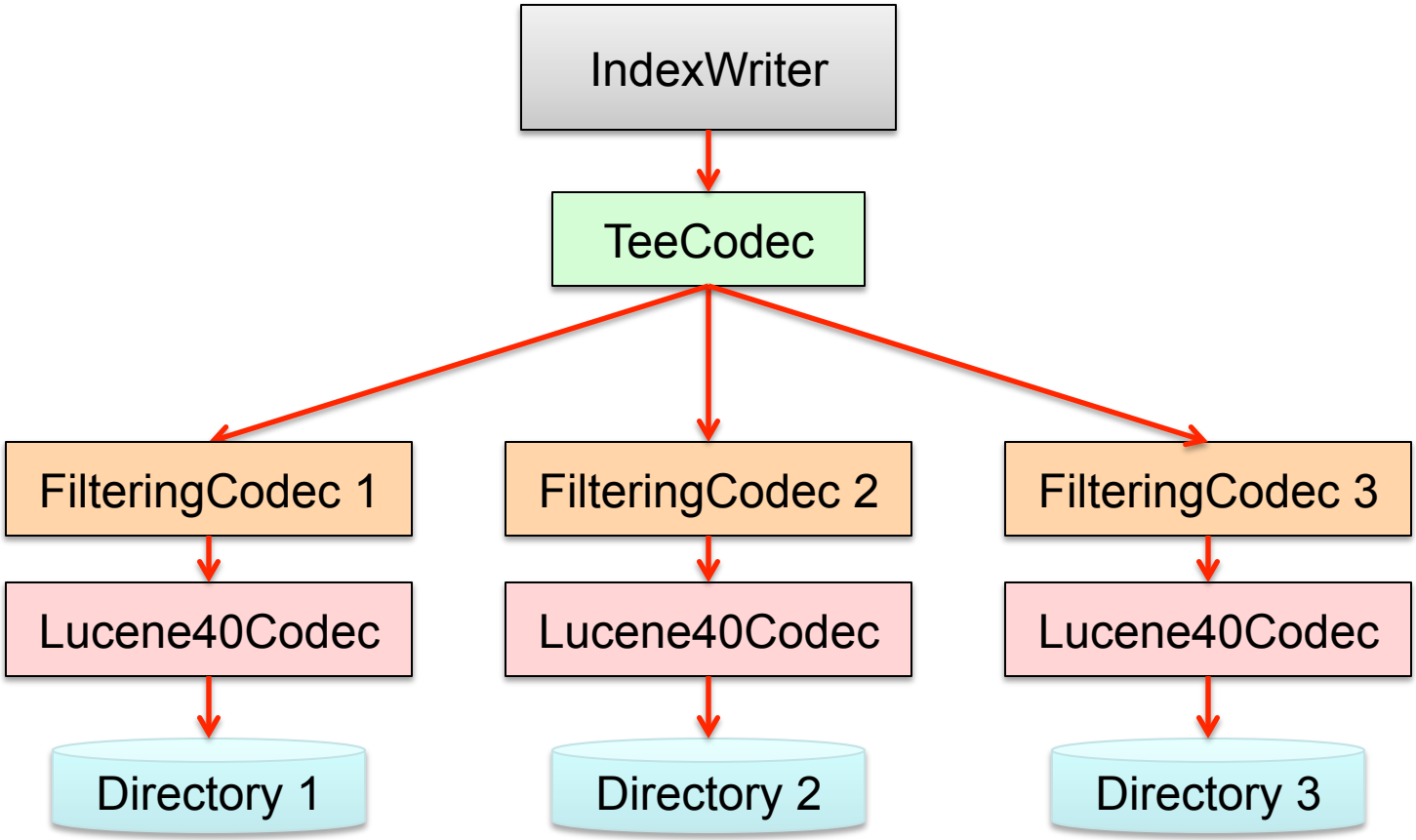
Example: index pruning

- On-the-fly pruning, i.e. no post-processing



Example: Single-pass IndexSplitter

- Each FilteringCodec selects a subset of data
  - **Not necessarily disjoint!**



Field-level index updates

Current index update design

- Document-level “update” is really a “delete + add”
  - **Old document ID* is hidden via “liveDocs” bitset**
  - **Term and collections statistics wrong for a time**
  - **Only a segment merge actually removes deleted document’s data (stored fields, postings, etc)**
    - *And fixes term / collection statistics*
  - **New document is added to a new segment, with a different ID***

* Internal document ID (segment scope) – ephemeral int, not preserved in segment merges

Problems with the current design

- Document-level
- Users have to store all fields
- All indexed fields have to be analyzed again
- Costly operation for large documents with small frequent updates
- Some workarounds exist:
  - **ParallelReader with large static index + small dynamic index – tricky to sync internals IDs!**
  - **ExternalFileField – simple float values, sorted in memory to match doc ID-s**
  - **Application-level join between indexes or index + db**

Let’s change it

“Stacked” field-level updates

- Per-field updates, both stored and inverted data
- Updated field data is “stacked” on top of old data
- Old data is “covered” by the updates
- Paper by Ercegovac, Josifovski, Li et al
  - **“Supporting Sub-Document Updates and Queries in an Inverted Index” CIKM ‘08**

Proposed “stacked” field updates

- Field updates represented as new documents
  - **Contain only updated field values**
    - *Additional stored field keeps the original doc ID? OR*
    - *Change & sort the ID-s to match the main segment?*
- Updates are written as separate segments
- On reading, data from the main and the “stacked” segments is *somehow* merged on the fly
  - **Internal ID-s have to be matched for the join**
    - *Original ID from the main index*
    - *Re-mapped, or identical ID from the stacked segment?*
  - **Older data replaced with the new data from the “stacked” segments**
- Re-use existing APIs when possible

NOTE: work in progress

- This is a work in progress
- Very early stage
- DO NOT expect this to work today – it doesn't!
  - **It's a car frame + a pile of loose parts**

Writing “stacked” updates

Writing “stacked” updates

- Updates are regular Lucene Document-s
  - **With the added “original ID” (oid) stored field**
  - **OR re-sort to match internal IDs of the main segment?**
- Initial design
  - **Additional IndexWriter-s / DocumentWriter-s – UpdateWriter-s**
  - **Create regular Lucene segments**
    - *E.g. using different namespace (`u_0f5` for updates of `_0f5`)*
  - **Flush needs to be synced with the main IndexWriter**
  - **SegmentInfos modified to record references to the update segments**
  - **Segment merging in main index closes UpdateWriter-s**
- Convenience methods in IndexWriter
  - **IW.updateDocument(int n, Document newFields)**
- End result: additional segment(s) containing updates

… to be continued …

- Interactions between the UpdateW and the main IW
- Support multiple stacked segments
- Evaluate strategies
  - **Map ID-s on reading, OR**
  - **Change & sort ID-s on write**
- Support NRT

Reading “stacked” updates

Combining updates with originals

- Updates may contain single or multiple fields
  - **Need to keep track what updated field is where**
- Multiple updates of the same document
  - **Last update should win**
- ID-s in the updates != ID-s in the main segment!
  - **Need a mapping structure between internal ID-s**
  - **OR: Sort updates so that ID-s match**
- ID mapping – costs to retrieve
- ID sorting – costs to create

* Initial simplification: max. 1 update segment for 1 main segment

Unsorted “stacked” updates

Runtime ID re-mapping

Unsorted updates – ID mismatch

- Resolve ID-s at runtime:
  - **Use stored original ID-s (newID → oldID)**
  - **Invert the relation and sort (oldID → newID)**
- Use a (sparse!) per-field map of oldID → newID for lookup and translation

  E.g. when iterating over docs:
  - **Foreach ID in old ID-s:**
    - *Check if oldID exists in updates*
    - *if exists, translate to newID and return the newID's data*

Stacked stored fields

Original segment

id	f1	f2
10	abba	c-b
11	b-ad	-b-c
12	ca--d	c-c
13	da-da	b--b

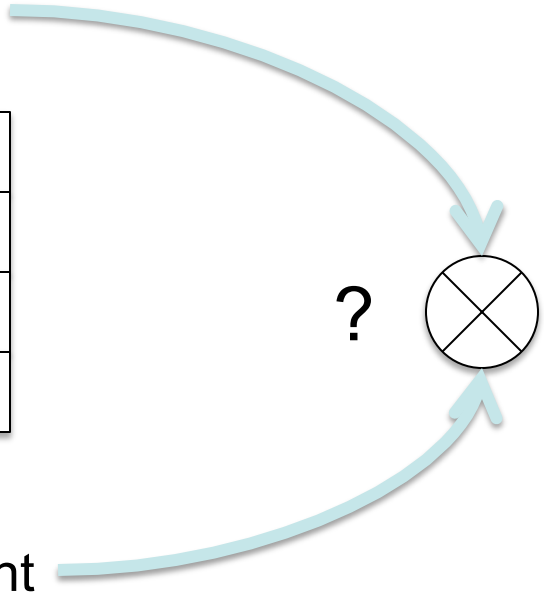
Funny looking field values? This is just to later illustrate the tokenization – one character becomes one token, and then it becomes one index term.

- Any non-inverted fields
  - Stored fields, norms or docValues

Stacked stored fields

Original segment

id	f1	f2
10	abba	c-b
11	b-ad	-b-c
12	ca--d	c-c
13	da-da	b--b



"Updates" segment

id	oid	f1	f2	f3
0	12	ba-a		
1	10	ac	--cb	
2	13			-ee
3	13	dab		
4	10	ad-c		

- Several versions of a field
- Fields spread over several updates (documents)
- Internal IDs don't match!
  - Store the original ID (oid)

Stacked stored fields

Original segment

id	f1	f2
10	abba	c-b
11	b-ad	-b-c
12	ca--d	c-c
13	da-da	b--b

- Build a map from original IDs to the IDs of updates
  - sort by oid
- One sparse map per field
- Latest field value wins
- Fast lookup needed

"Updates" segment

id	oid	f1	f2	f3
0	12	ba-a		
1	10	ac	--cb	
2	13			-ee
3	13	dab		
4	10	ad-c		

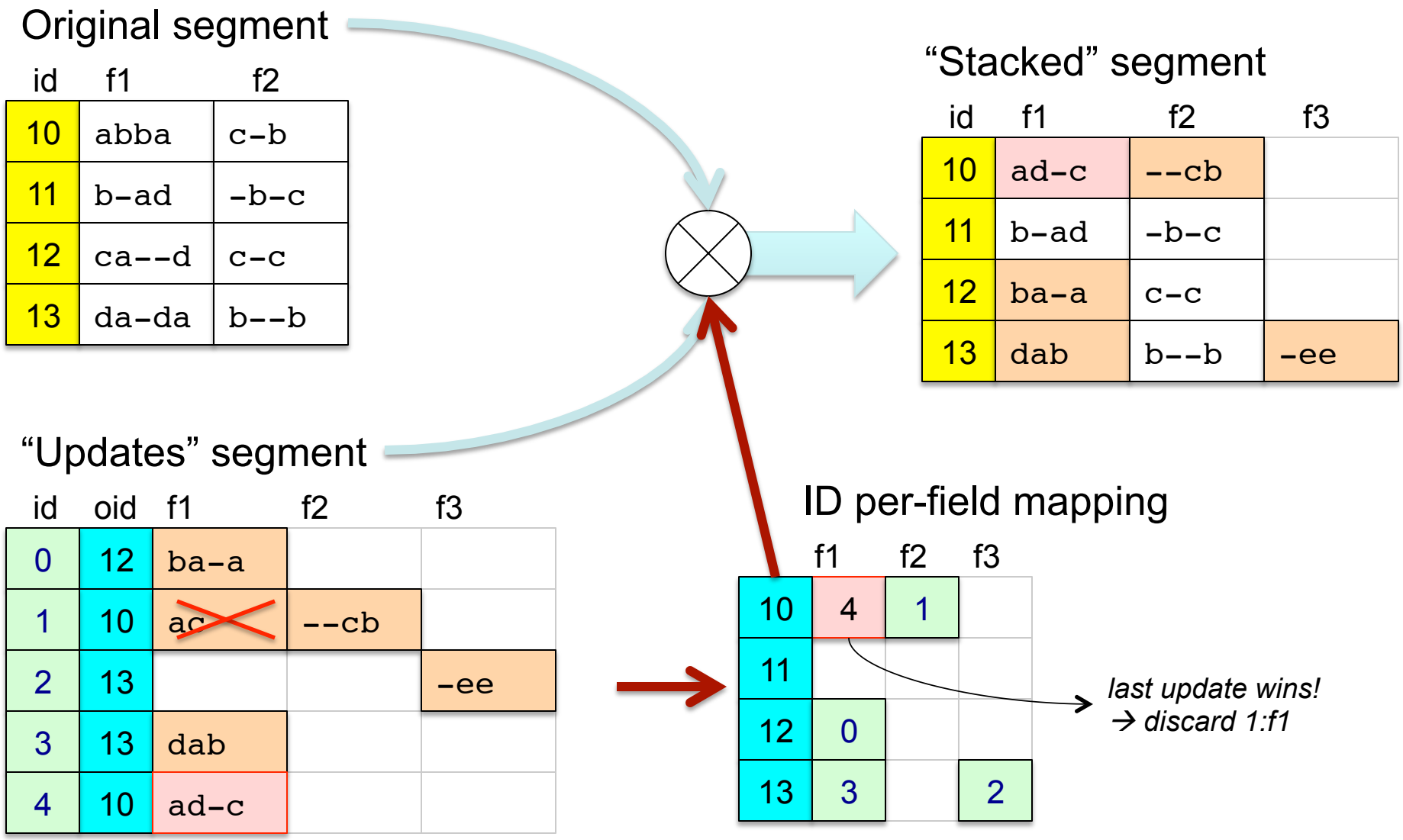
in memory?

ID per-field mapping

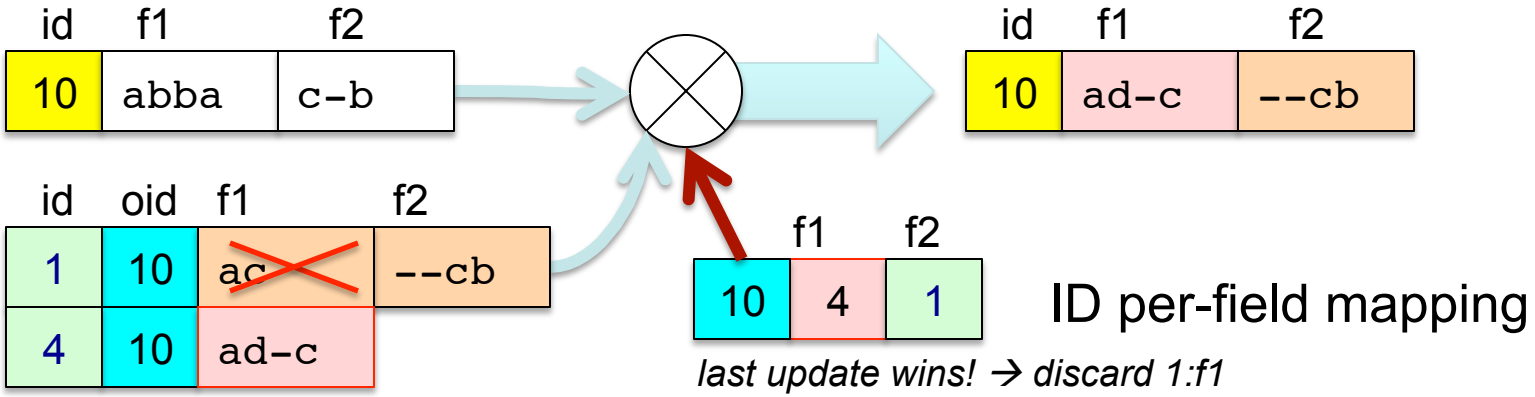
	f1	f2	f3
10	4	1	
11			
12	0		
13	3		2

last update wins!

Stacked stored fields



Stacked stored fields – lookup



- Initialize mapping table from the "updates" segment
  - **Doc 1 field1 (the first update of oid 10) is obsolete – discard**
- Get stored fields for doc 10:
  - **Check the mapping table what fields are updated**
  - **Retrieve field1 from doc 4 and field 2 from doc 1 in "updates"**

    NOTE: major cost of this approach - random seek!
  - **Retrieve any other original fields from the main segment for doc 10**
  - **Return a combined iterator of field values**

Stacked inverted fields

Original segment

	terms	id / postings 10	11	12	13
f1	a	0,3	2	1	1,4
	b	1,2	0		
	c			0	
	d		3	4	0,3
f2	b	2	1		1,3
	c	0	3	0,2	

inverted

```
10. f1: abba
    f2: c-b

11. f1: b-ad
    f2: -b-c

12. f1: ca--d
    f2: c-c

13. f1: da-da
    f2: b--b
```

- Inverted fields have:
  - **Fields**
  - **Term dictionary + term freqs**
  - **Document frequencies**
  - **Positions**
  - **Attributes (offsets, payloads, …)**
    - *…and norms, but norms are non-inverted == like stored fields*
- Updates should overlay “cells” for each term at <field,term,doc>
  - **Positions, attributes**
  - **Discard all old data from the cell**

Stacked inverted fields

Original segment

	terms	id / postings 10	11	12	13
f1	a	0,3	2	1	1,4
	b	1,2	0		
	c			0	
	d		3	4	0,3
f2	b	2	1		1,3
	c	0	3	0,2	

"Updates" segment

	terms	*12* 0	*10* 1	*13* 2	*13* 3	*10* 4
f1	a	1,3	0		1	0
	b	0			2	
	c		1			3
	d				0	1
f2	b		3			
	c		2			
f3	e			1,2		

Documents containing updates of inverted fields:

```
0. f1: ba-a     (oid: 12)

1. f1: ac       (oid: 10)
   f2: --cb

2. f3: -ee      (oid: 13)

3. f1: dab      (oid: 13)

4. f1: ad-c     (oid: 10)
```

Stacked inverted fields

Original segment

id / postings

terms		10	11	12	13
f1	a	0,3	2	1	1,4
	b	1,2	0		
	c			0	
	d		3	4	0,3
f2	b	2	1		1,3
	c	0	3	0,2	

"Updates" segment

terms		*12*	*10*	*13*	*13*	*10*
		0	1	2	3	4
f1	a	1,3	0		1	0
	b	0			2	
	c		1			3
	d				0	1
f2	b		3			
	c		2			
f3	e			1,2		

- ID mapping table:
  - **The same sparse table!**
  - **Take the latest postings at the new doc ID**
  - **Ignore original postings at the original doc ID**

ID per-field mapping

	f1	f2	f3
10	4	1	
11			
12	0		
13	3		2

last update wins!
→ discard 1:f1

Stacked inverted fields

Original segment

	terms	id / postings 10	11	12	13
f1	a	0,3	2	1	1,4
	b	1,2	0		
	c			0	
	d		3	4	0,3
f2	b	2	1		1,3
	c	0	3	0,2	

"Updates" segment

	terms	*12* 0	*10* 1	*13* 2	*13* 3	*10* 4
f1	a	1,3	~~3~~		1	0
	b	0	~~0~~		2	
	c		~~1~~			3
	d		~~2~~		0	1
f2	b		3			
	c		2			
f3	e			1,2		

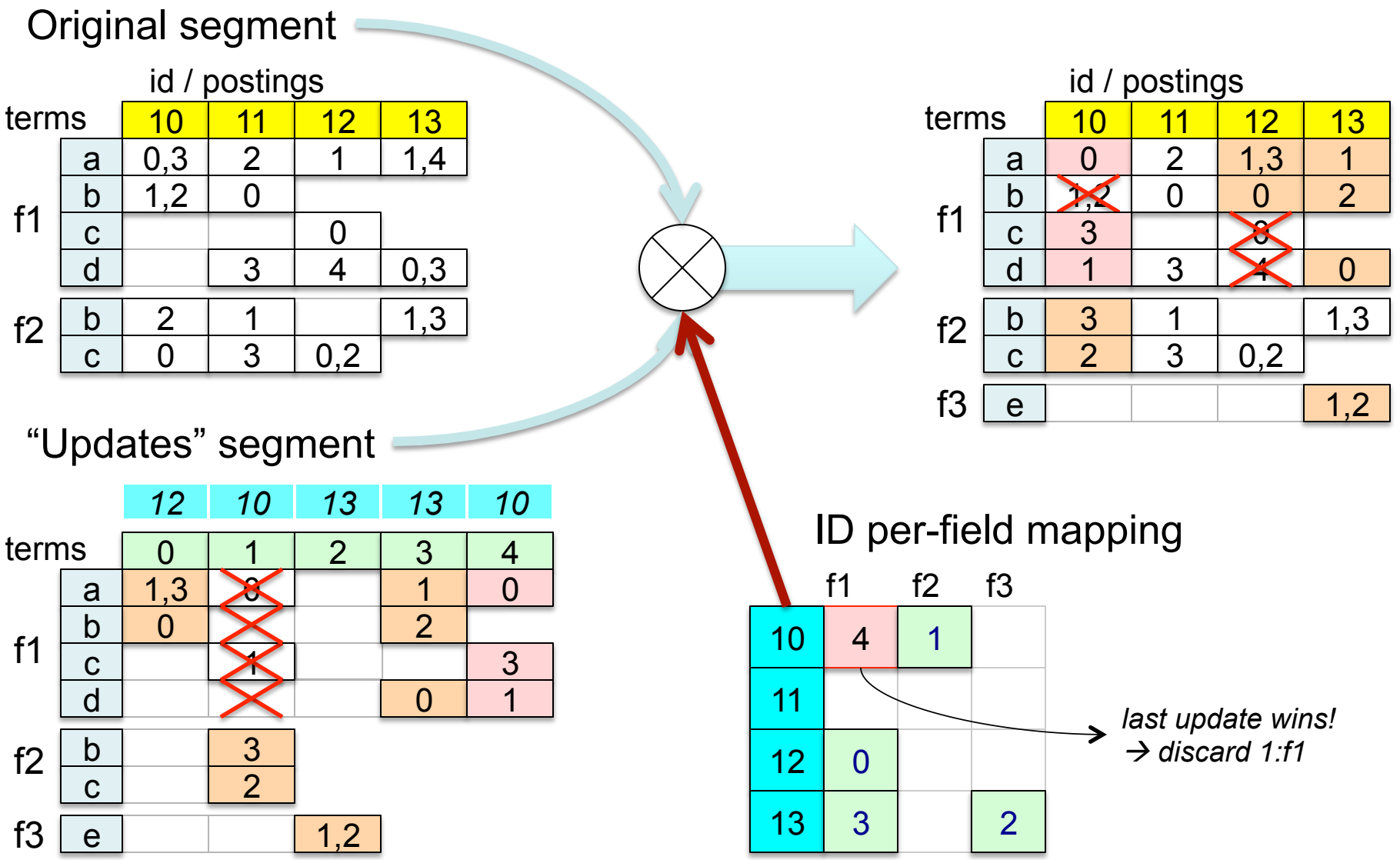
- ID mapping table:
  - **The same sparse table!**
  - **Take the latest postings at the new doc ID**
  - **Ignore original postings at the original doc ID**

ID per-field mapping

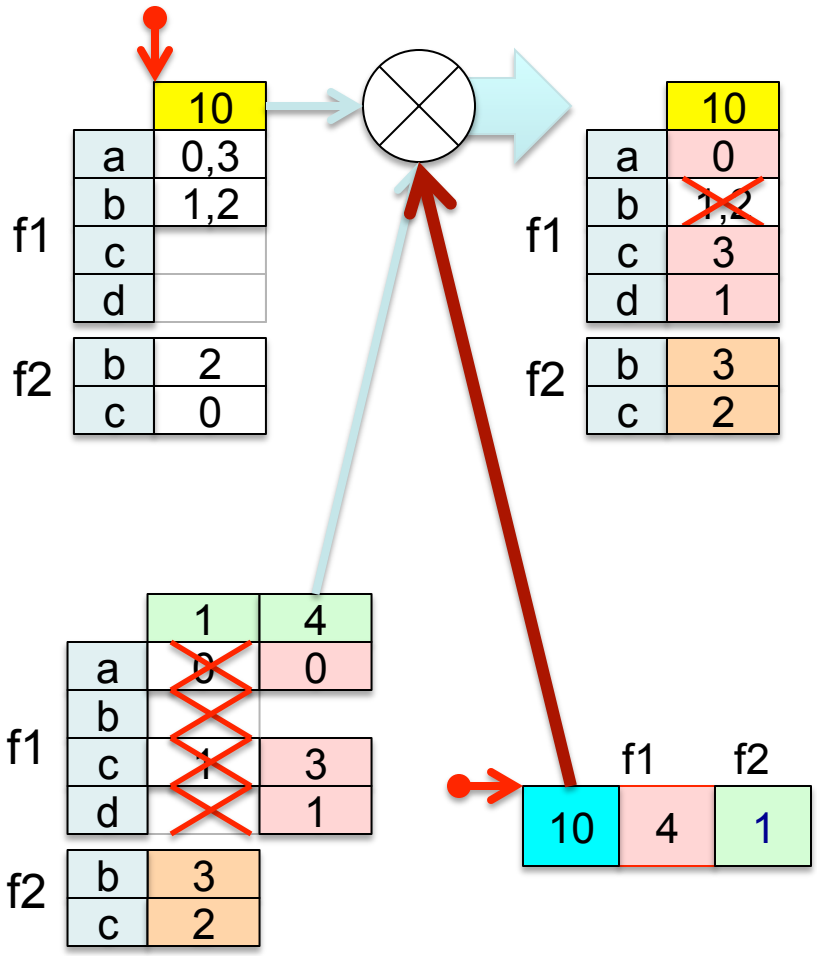
	f1	f2	f3
10	4	1	
11			
12	0		
13	3		2

last update wins!
→ discard 1:f1

Stacked inverted fields



Stacked inverted fields – lookup



TermsEnum and DocsEnum need a merged list of terms and a merged list of id-s per term

- Re-use mapping table for the "updates" segment
- Iterate over posting list for "f1:a"
  - **Check both lists!**
  - **ID 10: present in the mappings, discard original in-doc postings**
    - *ID not present in the mappings → return original in-doc postings*
  - **Retrieve new postings from <f1,a,doc4> in "updates"**

    NOTE: major cost – random seek!
  - **Advance to the next doc ID**

Implementation details

- SegmentInfos extended to keep names of “stacked” segments
  - **“Stacked” segments in a different namespace**
- Stacked Codec *Producers that combine & remap data
- SegmentReader/SegmentCoreReaders modified
  - **Check for and open a “stacked” SegmentReader**
  - **Read and construct the ID mapping table**
  - **Create stacked Codec *Producers initialized with:**
    - *Original format *Producers*
    - *Stacked format *Producers*
    - *The ID mapping table*

Merged fields

- Field lists merge easily
  - **Trivial, very little data to cache & merge**
- StoredFieldsProducer merges easily
- However, TermsEnum and DocsEnum enumerators need more complex handling …

Leapfrog enumerators

- Terms and postings have to be merged
  - **But we don't want to fully read all data!**
- Use "leapfrog" enumeration instead
  - **INIT: advance both main and stacked enum**
  - **Return from the smaller, and keep advancing & returning from the smaller until it reaches (or exceeds) the current value from the larger**
  - **If values are equal then merge the data – again, in a leapfrog fashion; advance both**
    - *Similar to MultiTermsEnum but simpler*

Segment merging

- Merging segments with “stacked” updates is trivial because …
  - **All Codec enumerators already present a unified view of data!**
- Just delete both the main and the “stacked” segment after a merge is completed
  - **Updates are already rolled in into the new segment**

Limitations

- Search-time costs
  - **Mapping table consumes memory**
  - **Overheads of merging postings and field values**
  - **Many random seeks in "stacked" segments due to oldID → newID**
- Trade-offs
  - **Performance impact minimized if this data is completely in memory → fast seek**
  - **Memory consumption minimized if this data is on-disk → slow seek**
  - **Conclusion: size of updates should be kept small**
- Difficult to implement Near-Real-Time updates?
  - **Mapping table incr. updates, not full rebuilds**

… to be continued …

- Evaluate the cost of runtime re-mapping of ID-s and random seeking
- Extend the design to support multi-segment stacks
- Handle deletion of fields

Current status

- LUCENE-3837
- Branch in Subversion – lucene3837
- Very early stage – experiments
- Initial code for StackedCodec formats and SegmentReader modifications
- Help needed!

Summary & QA

- Codec API in Lucene 4
- Some Codec applications: tee, filtering, splitting

  http://issues.apache.org/jira/browse/LUCENE-2632
- Field-level index updates
  - **"Stacked" design, using adjacent segments**
  - **ID mapping table**
- Help needed!

  http://issues.apache.org/jira/browse/LUCENE-3837

- More questions? ab@lucidimagination.com

Bonus slides

TeeDirectory

- Makes literal copies of Directory data
  - **As it's being created, byte by byte**
- Simple API:

  Directory out = new TeeDirectory(main, others…);
- Can exclude some files from copying, by prefix
  - **E.g. "_0" – exclude all files of segment _0**
- Can perform initial sync
  - **Bulk copy from existing main directory to copies**
- Mirroring on the fly – more fine-grained than commit-based replication
  - **Quicker convergence of copies with the main dir**

Sorted “stacked” updates

Changing and syncing ID-s on each update

(briefly)

Sorted updates

- Essentially the ParallelReader approach
  - **Requires synchronized ID-s between segments**
  - **Some data structures need "fillers" for absent ID-s**
- Updates arrive out of order
  - **Updates initially get unsynced ID-s**
- On flush of the segment with updates
  - **Multiple updates have to be collapsed into single documents**
  - **ID-s have to be remapped**
  - **The "updates" segment has to be re-written**
    - *LUCENE-2482 Index sorter – possible implementation*

Reading sorted updates

- A variant of ParallelReader
  - **If data is present both in the main and in the secondary indexes, return the secondary data and drop the main data**
- Nearly no loss of performance or memory!
- But requires re-building and sorting (rewrite) of the secondary segment on every update ☹

- LUCENE-3837 uses the “unsorted” design, with the ID mapping table and runtime re-mapping