Is Your IndexReader Really Atomic or Maybe Slow?

Uwe Schindler

SD DataSolutions GmbH,
uschindler@sd-datasolutions.de

My Background

- I am committer and PMC member of Apache Lucene and Solr. My main focus is on development of Lucene Java.
- Implemented fast numerical search and maintaining the new attribute-based text analysis API. Well known as Generics and Sophisticated Backwards Compatibility Policeman.
- Working as consultant and software architect for SD DataSolutions GmbH in Bremen, Germany. The main task is maintaining PANGAEA (Publishing Network for Geoscientific & Environmental Data) where I implemented the portal's geo-spatial retrieval functions with Apache Lucene Core.
- Talks about Lucene at various international conferences like the previous Berlin Buzzwords, Lucene Revolution, Lucene Eurocon, ApacheCon EU/US, and various local meetups.

Agenda

- Motivation / History of Lucene
- AtomicReader & CompositeReader
- Reader contexts
- Wrap up

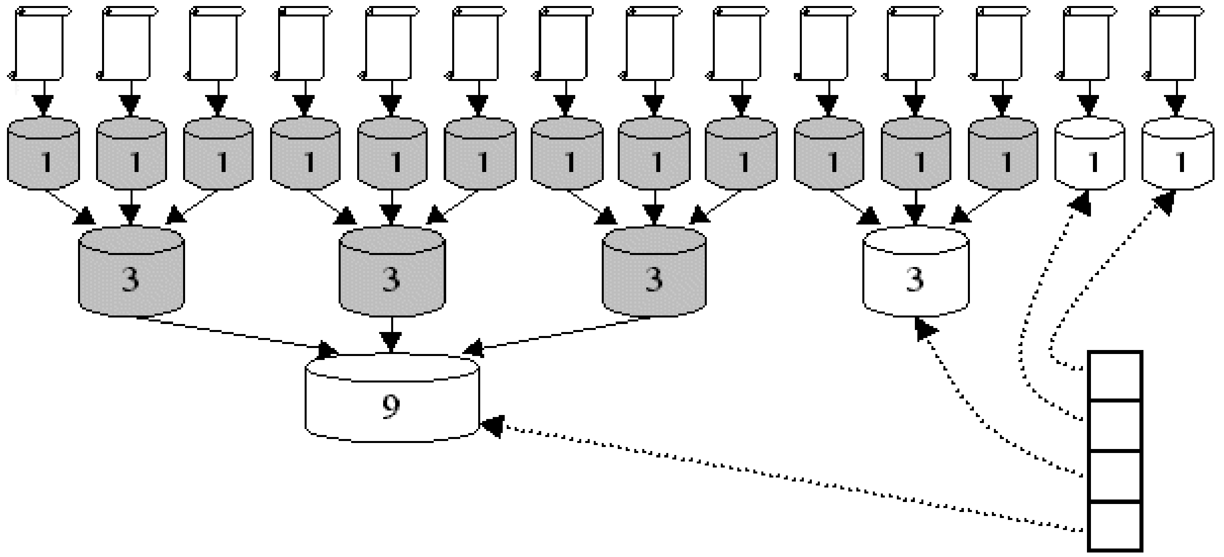
Lucene Index Structure

- Lucene was the first full text search engine that supported **document additions and updates**
- **Snapshot isolation** ensures consistency

$\Rightarrow$ Segmented index structure

$\Rightarrow$ Committing changes creates new segments

Segments in Lucene

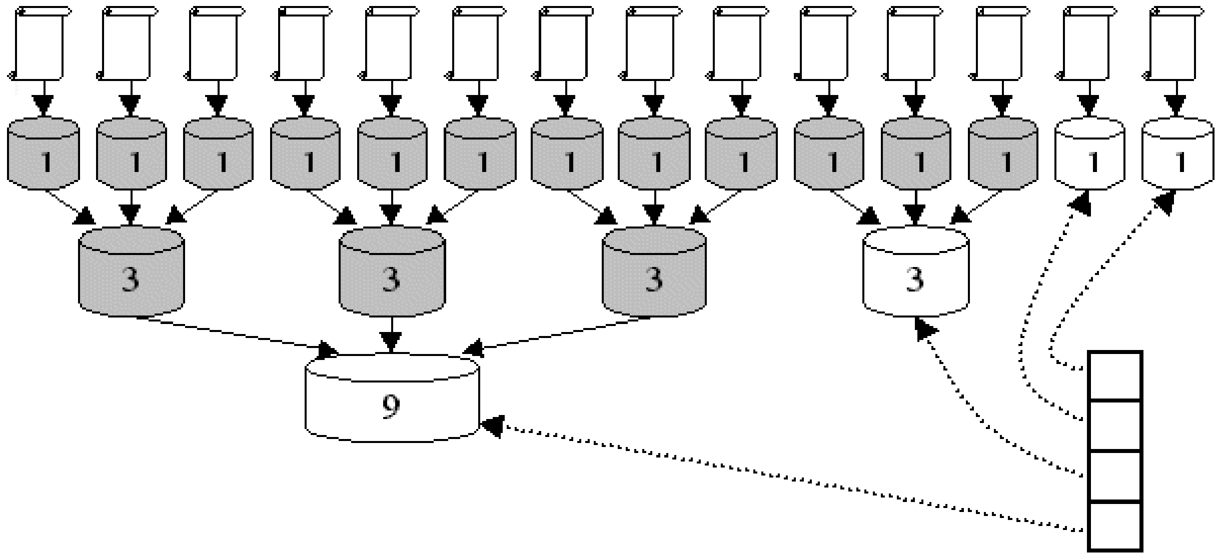


- Each index consists of various segments placed in the index directory. All documents are added to new new segment files, merged with other on-disk files after flushing.

Image: Doug Cutting

*) The term “optimal” does not mean all indexes must be optimized!

Segments in Lucene

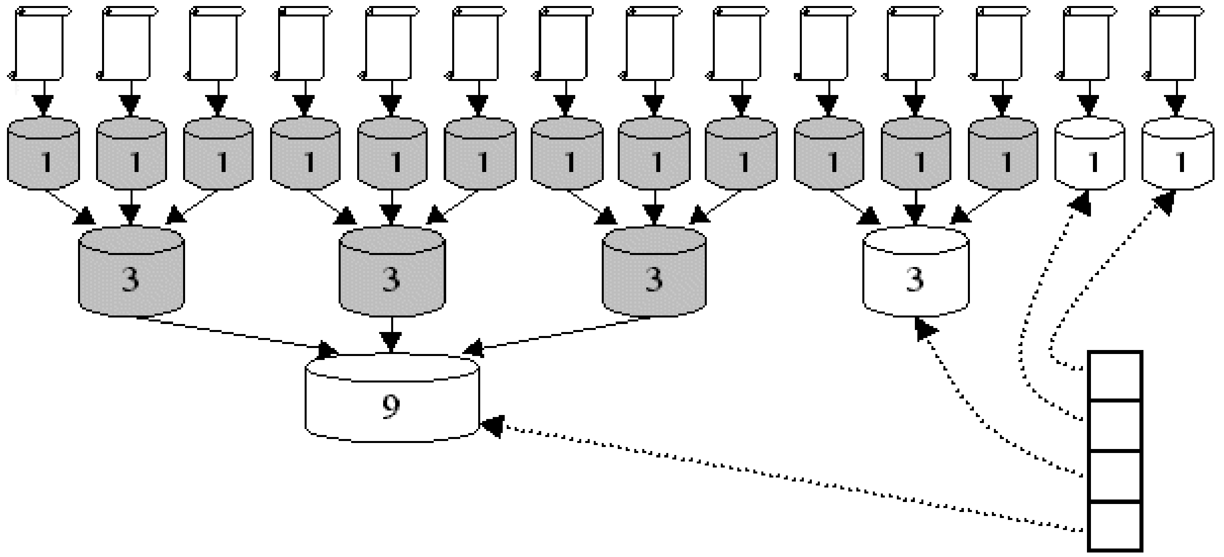


- Each index consists of various segments placed in the index directory. All documents are added to new new segment files, merged with other on-disk files after flushing.
- Lucene writes segments incrementally and can merge them.

Image: Doug Cutting

*) The term “optimal” does not mean all indexes must be optimized!

Segments in Lucene

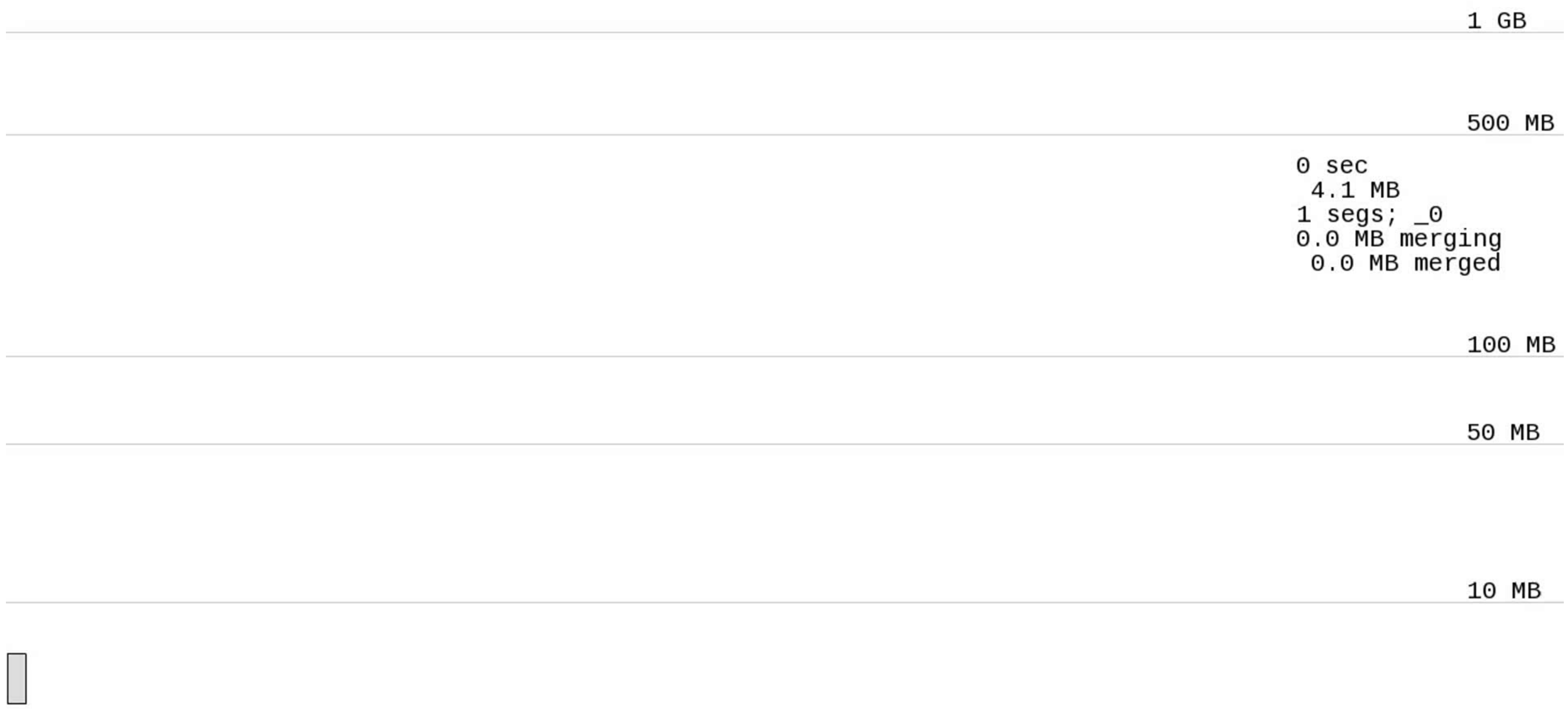


- Each index consists of various segments placed in the index directory. All documents are added to new new segment files, merged with other on-disk files after flushing.
- Lucene writes segments incrementally and can merge them.
- *Optimized** index consists of one segment.

*) The term “optimal” does not mean all indexes must be optimized!

Image: Doug Cutting

Lucene merges while indexing all of English Wikipedia



Video by courtesy of:
Mike McCandless' blog, http://goo.gl/kI53f

Indexes in Lucene *(up to version 3.6)*

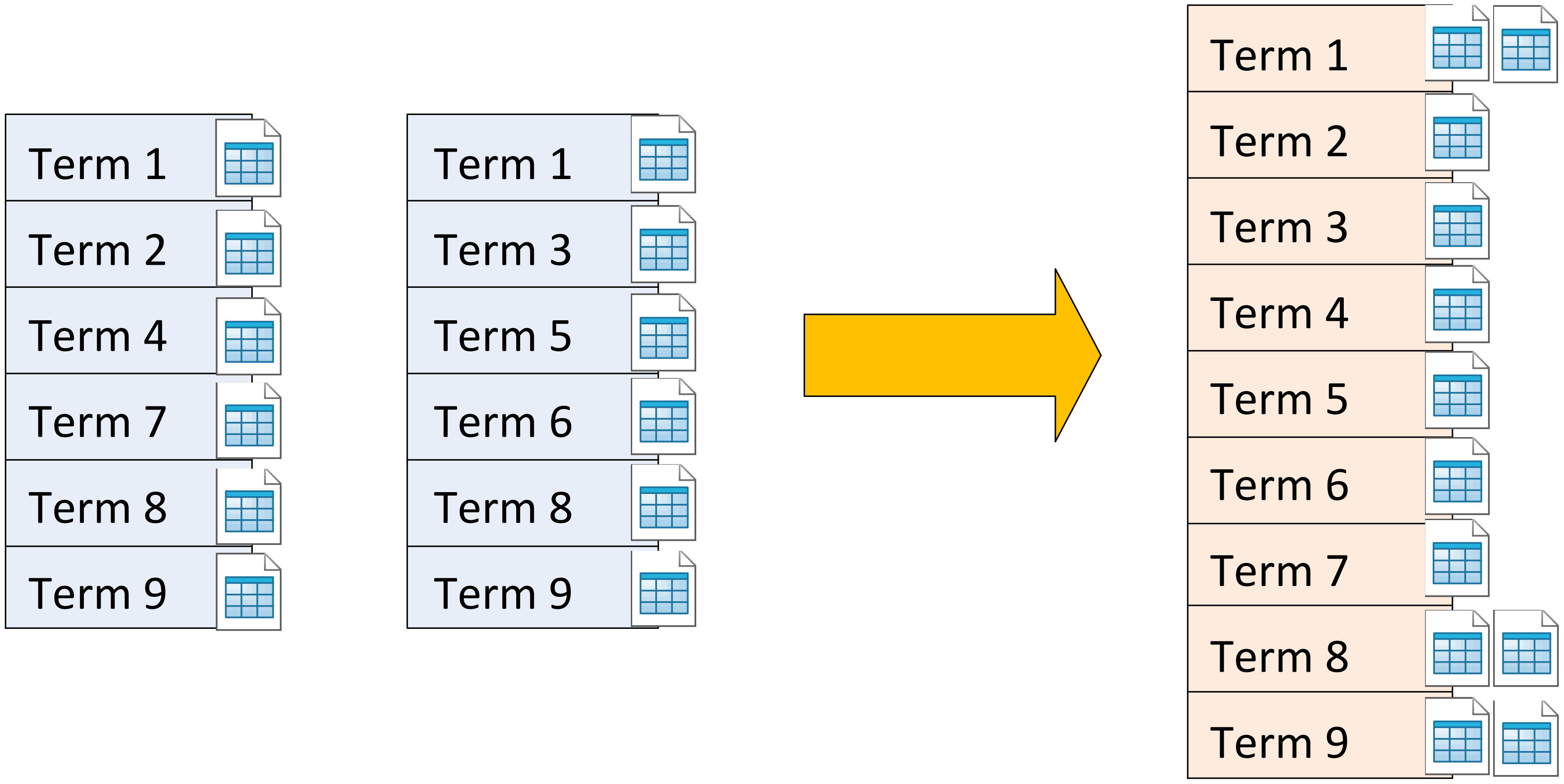
- Each segment (“atomic index”) is a completely functional index:
  - SegmentReader implements the IndexReader interface for single segments
- Composite indexes
  - DirectoryReader implements the IndexReader interface on top of a set of SegmentReaders
  - MultiReader is an abstraction of multiple IndexReaders combined to one virtual index

Composite Indexes *(up to version 3.6)*

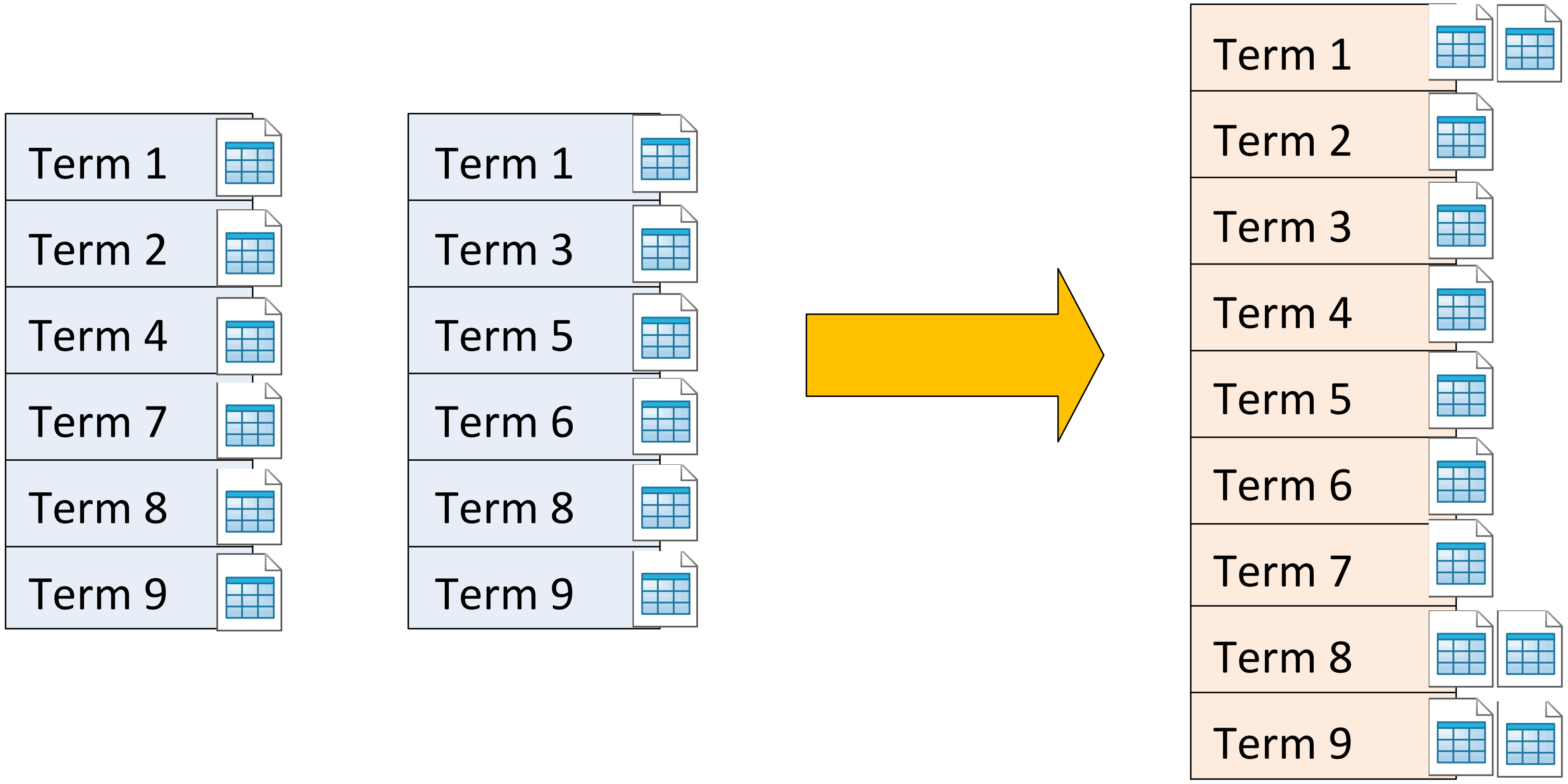
Atomic “views” on multi-segment index:

- **Term Dictionary:** on-the-fly merged & sorted term index (priority queue for TermEnum,…)
- **Postings:** posting lists for each term appended, convert document IDs to be global

Merging Term Index and Postings



Merging Term Index and Postings



Composite Indexes *(up to version 3.6)*

Atomic "views" on multi-segment index:

- **Term Dictionary:** on-the-fly merged & sorted term index (priority queue for TermEnum,…)
- **Postings:** posting lists for each term appended, convert document IDs to be global
- **Metadata:** doc frequency, term frequency,…
- **Stored fields, term vectors, deletions:** delegate global document IDs -> segment document IDs (binary search)
- **FieldCache:** duplicate instances for single segments and composite view (memory!!!)

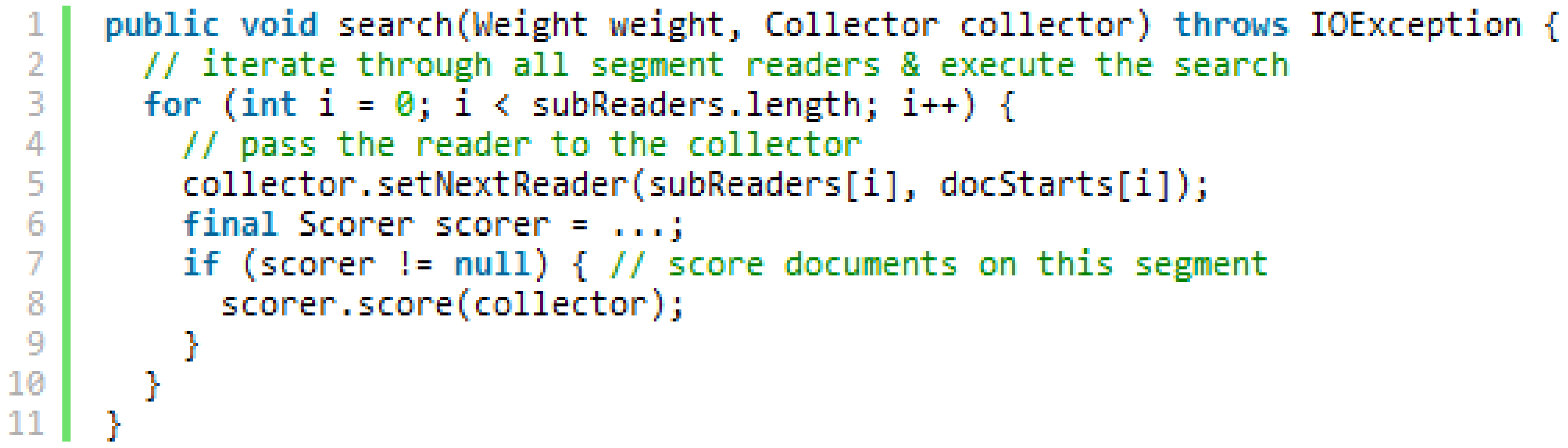
Searching before Version 2.9

- IndexSearcher used the underlying index always as a “single” *(atomic)* index:
  - Queries are executed on the atomic view of a composite index
  - Slowdown for queries that scan term dictionary (MultiTermQuery) or hit lots of documents, facetting
    => recommendation to “optimize” index
  - On every index change, FieldCache used for sorting had to be reloaded completely

Image: © The Walt Disney Company

Lucene 2.9 and later:
Per segment search

Search is executed separately on each index segment:



```java
public void search(Weight weight, Collector collector) throws IOException {
  // iterate through all segment readers & execute the search
  for (int i = 0; i < subReaders.length; i++) {
    // pass the reader to the collector
    collector.setNextReader(subReaders[i], docStarts[i]);
    final Scorer scorer = ...;
    if (scorer != null) { // score documents on this segment
      scorer.score(collector);
    }
  }
}
```

Atomic view no longer used!

Per Segment Search: Pros

- No live term dictionary merging
- Possibility to parallelize
  - ExecutorService in IndexSearcher since Lucene 3.1+
  - **Do not optimize to make this work!**
- Sorting only needs per-segment FieldCache
  - Cheap reopen after index changes!
- Filter cache per segment
  - Cheap reopen after index changes!

Per Segment Search: Cons

- Query/Filter API changes:
  - Scorer / Filter’s DocIdSet no longer use global document IDs
- Slower sorting by string terms
  - Term ords are only comparable inside each segment
  - String comparisons needed after segment traversal
  - **Use numeric sorting if possible!!!**
    (Lucene supports missing values since version 3.4 [buggy], corrected 3.5+)

Agenda

- Motivation / History of Lucene
- AtomicReader & CompositeReader
- Reader contexts
- Wrap up

Composite Indexes *(up to version 3.6)*

Atomic "views" on multi-segment index:

- **Term Dictionary:** on-the-fly merged & sorted term index (priority queue for TermEnum,…)
- **Postings:** posting lists for each term appended, convert document IDs to be global
- **Metadata:** doc frequency, term frequency,…
- **Stored fields, term vectors, deletions:** delegate global document IDs -> segment document IDs (binary search)
- **FieldCache:** duplicate instances for single segments and composite view (memory!!!)

Composite Indexes *(version 4.0)*

~~Atomic~~ **“views” on multi-segment index:**

- ~~**Term Dictionary:** on-the-fly merged & sorted term index (priority queue for TermEnum,...)~~
- ~~**Postings:** posting lists for each term appended, convert document IDs to be global~~
- **Metadata:** doc frequency, ~~term frequency~~,...
- **Stored fields, term vectors, ~~deletions~~:** delegate global document IDs -> segment document IDs (binary search)
- ~~**FieldCache:** duplicate instances for single segments and composite view (memory!!!)~~

Early Lucene 4.0

- Only “historic” IndexReader interface available since Lucene 1.0
- **80%** of all methods of <u>composite</u> IndexReaders throwed **UnsupportedOperationException**
  - This affected all user-facing APIs (SegmentReader was hidden / marked experimental)
- **No compile time safety!**
  - Query Scorers and Filters need term dictionary and postings, throwing UOE when executed on composite reader

Heavy Committing™ !!!

IndexReader

- Most generic (abstract) API to an index
  - superclass of more specific types
  - cannot be subclassed directly *(no public constructor)*

IndexReader

- Most generic (abstract) API to an index
  - superclass of more specific types
  - cannot be subclassed directly *(no public constructor)*
- Does not know anything about “inverted index” concept
  - no terms, no postings!

IndexReader

- Most generic (abstract) API to an index
  - superclass of more specific types
  - cannot be subclassed directly *(no public constructor)*
- Does not know anything about “inverted index” concept
  - no terms, no postings!
- Access to **stored fields** (and term vectors) by document ID
  - to display / highlight search results only!

IndexReader

- Most generic (abstract) API to an index
  - superclass of more specific types
  - cannot be subclassed directly *(no public constructor)*
- Does not know anything about “inverted index” concept
  - no terms, no postings!
- Access to **stored fields** (and term vectors) by document ID
  - to display / highlight search results only!
- Some very limited metadata
  - number of documents,…

IndexReader

- Most generic (abstract) API to an index
  - superclass of more specific types
  - cannot be subclassed directly *(no public constructor)*
- Does not know anything about “inverted index” concept
  - no terms, no postings!
- Access to **stored fields** (and term vectors) by document ID
  - to display / highlight search results only!
- Some very limited metadata
  - number of documents,…
- **Unmodifiable:** No more deletions / changing of norms possible!
  - **Applies to all readers in Lucene 4.0 !!!**

IndexReader

- Most generic (abstract) API to an index
  - superclass of more specific types
  - cannot be subclassed directly *(no public constructor)*
- Does not know anything about "inverted index" concept
  - no terms, no postings!
- Access to **stored fields** (and term vectors) by document ID
  - to display / highlight search results only!
- Some very limited metadata
  - number of documents,…
- **Unmodifiable:** No more deletions / changing of norms possible!
  - **Applies to all readers in Lucene 4.0 !!!**

Passed to IndexSearcher

just as before!

IndexReader

- Most generic (abstract) API to an index
  - superclass of more specific types
  - cannot be subclassed directly *(no public constructor)*
- Does not know anything about “inverted index” concept
  - no terms, no postings!
- Access to **stored fields** (and term vectors) by document ID
  - to display / highlight search results only!
- Some very limited metadata
  - number of documents,…
- **Unmodifiable:** No more deletions / changing of norms possible!
  - **Applies to all readers in Lucene 4.0 !!!**

Passed to IndexSearcher

just as before!

- Allows **IndexReader.open()** for backwards compatibility *(deprecated)*

AtomicReader

- Inherits from IndexReader

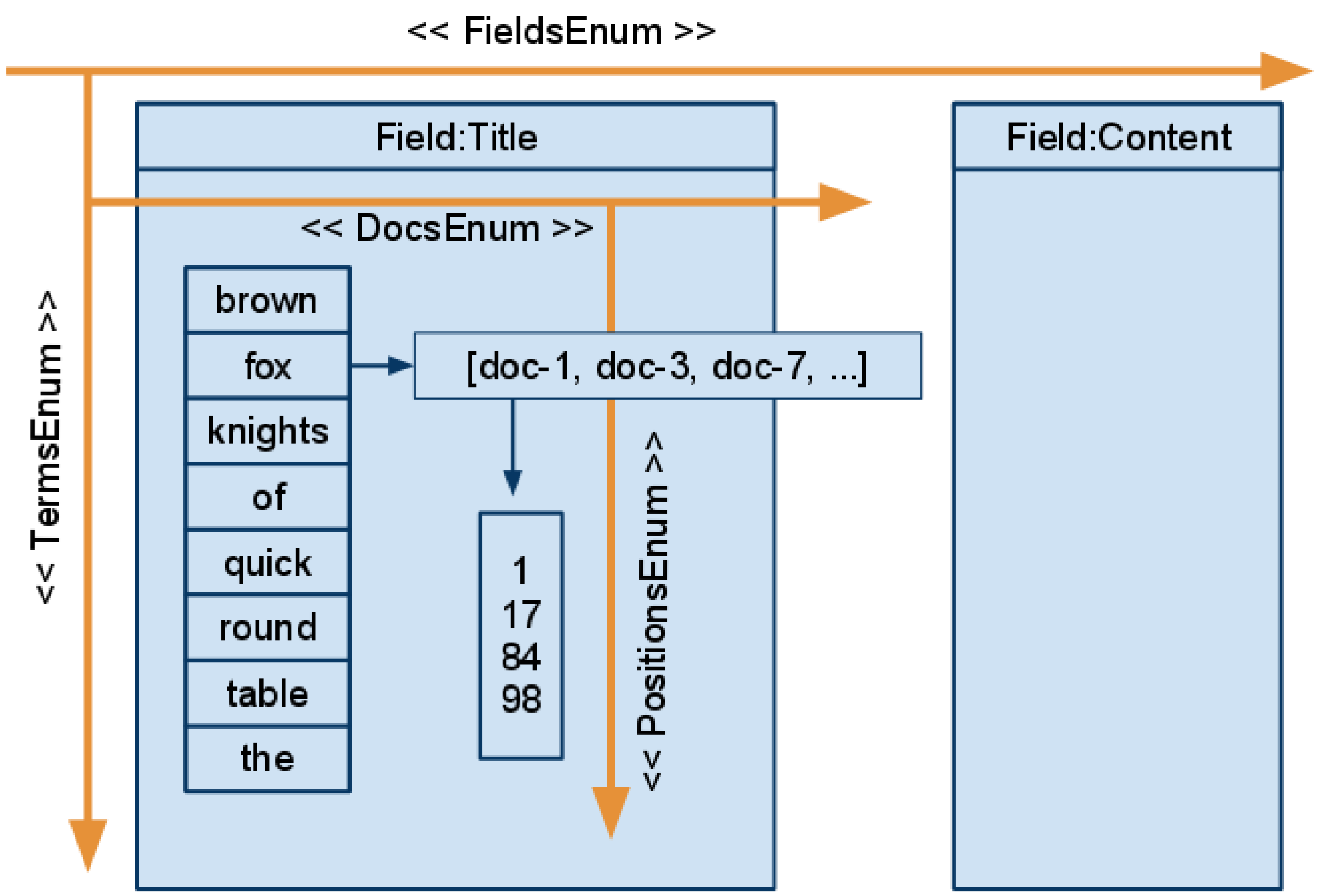
AtomicReader

- Inherits from IndexReader
- Access to “atomic” indexes (single segments)

AtomicReader

- Inherits from IndexReader
- Access to “atomic” indexes (single segments)
- Full term dictionary and postings API

AtomicReader



AtomicReader

- Inherits from IndexReader
- Access to “atomic” indexes (single segments)
- Full term dictionary and postings API
- Access to DocValues *(new in Lucene 4.0)* and norms

CompositeReader

- **No additional functionality** on top of IndexReader

CompositeReader

- **No additional functionality** on top of IndexReader
- Provides **getSequentialSubReaders()** to retrieve all child readers

CompositeReader

- **No additional functionality** on top of IndexReader
- Provides **getSequentialSubReaders()** to retrieve all child readers
- **DirectoryReader** and **MultiReader** implement this class

DirectoryReader

- Abstract class, defines interface for:

DirectoryReader

- Abstract class, defines interface for:
  - access to on-disk indexes *(on top of Directory class)*
  - access to **commit points, index metadata, index version, isCurrent()** for reopen support
  - defines abstract **openIfChanged** (for cheap reopening of indexes)
  - child readers are always **AtomicReader** instances

DirectoryReader

- Abstract class, defines interface for:
  - access to on-disk indexes *(on top of Directory class)*
  - access to **commit points, index metadata, index version, isCurrent()** for reopen support
  - defines abstract **openIfChanged** (for cheap reopening of indexes)
  - child readers are always **AtomicReader** instances
- Provides static factory methods for opening indexes
  - well-known from IndexReader in Lucene 1 to 3
  - factories return internal DirectoryReader implementation (StandardDirectoryReader with SegmentReaders as childs)

Basic Search Example

```
DirectoryReader reader = DirectoryReader.open(directory);
IndexSearcher searcher = new IndexSearcher(reader);
Query query = new QueryParser("fieldname", analyzer).parse(“text”);
TopDocs hits = searcher.search(query, 10);
ScoreDoc[] docs = hits.scoreDocs;
Document doc1 = searcher.doc(docs[0].doc);
// alternative:
Document doc2 = reader.document(docs[1].doc);
```

Image: © The Walt Disney Company

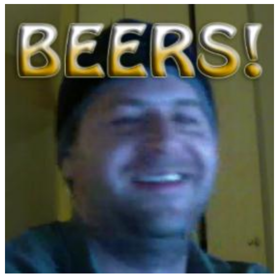
Looks familiar, doesn’t it?

*“**Arrgh!** I still need terms and postings on my DirectoryReader!!! What should I do? <u>Optimize</u> to only have one segment? **Help me please!!!**”*

(example question on java-user@lucene.apache.org)

What to do?

- Calm down
- Take a break and drink a beer*
- **Don't optimize** / force merge your index!!!



*) Robert's solution

Solutions

- **Most efficient way:**
  - Retrieve atomic leaves from your composite: `reader.getTopReaderContext().leaves()`
  - Iterate over sub-readers, do the work *(possibly parallelized)*
  - Merge results

Solutions

- **Most efficient way:**
  - Retrieve atomic leaves from your composite: `reader.getTopReaderContext().leaves()`
  - Iterate over sub-readers, do the work *(possibly parallelized)*
  - Merge results

- Otherwise: *wrap your CompositeReader:*

(Slow) **Solution**

```
AtomicReader
  SlowCompositeReaderWrapper.wrap(IndexReader r)
```

- **Wraps IndexReaders** of any kind as atomic reader, providing terms, postings, deletions, doc values
  - Internally uses same algorithms like previous Lucene readers
  - Segment-merging uses this to merge segments, too

(Slow) **Solution**

```
AtomicReader
  SlowCompositeReaderWrapper.wrap(IndexReader r)
```

- **Wraps IndexReaders** of any kind as atomic reader, providing terms, postings, deletions, doc values
  - Internally uses same algorithms like previous Lucene readers
  - Segment-merging uses this to merge segments, too

- Solr always provides an AtomicReader for convenience through **SolrIndexSearcher**. Plugin writers should use:

```
AtomicReader
  rd = mySolrIndexSearcher.getAtomicReader()
```

Other readers

- **FilterAtomicReader**
  - Was FilterIndexReader in 3.x
    (but now solely works on atomic readers)
  - Allows to filter terms, postings, deletions
  - Useful for index splitters (e.g., PKIndexSplitter, MultiPassIndexSplitter)
    (provide own getLiveDocs() method, merge to IndexWriter)
- **ParallelAtomicReader**, -**CompositeReader**

IndexReader Reopening

- Reopening **solely** provided by directory-based **DirectoryReader** instances

IndexReader Reopening

- Reopening **solely** provided by directory-based **DirectoryReader** instances
- No more reopen for:
  - **AtomicReader:** they are atomic, no refresh possible
  - **MultiReader:** reopen child readers separately, create new MultiReader on top of reopened readers
  - **Parallel*Reader, FilterAtomicReader:** reopen wrapped readers, create new wrapper afterwards

Agenda

- Motivation / History of Lucene
- AtomicReader & CompositeReader
- Reader contexts
- Wrap up

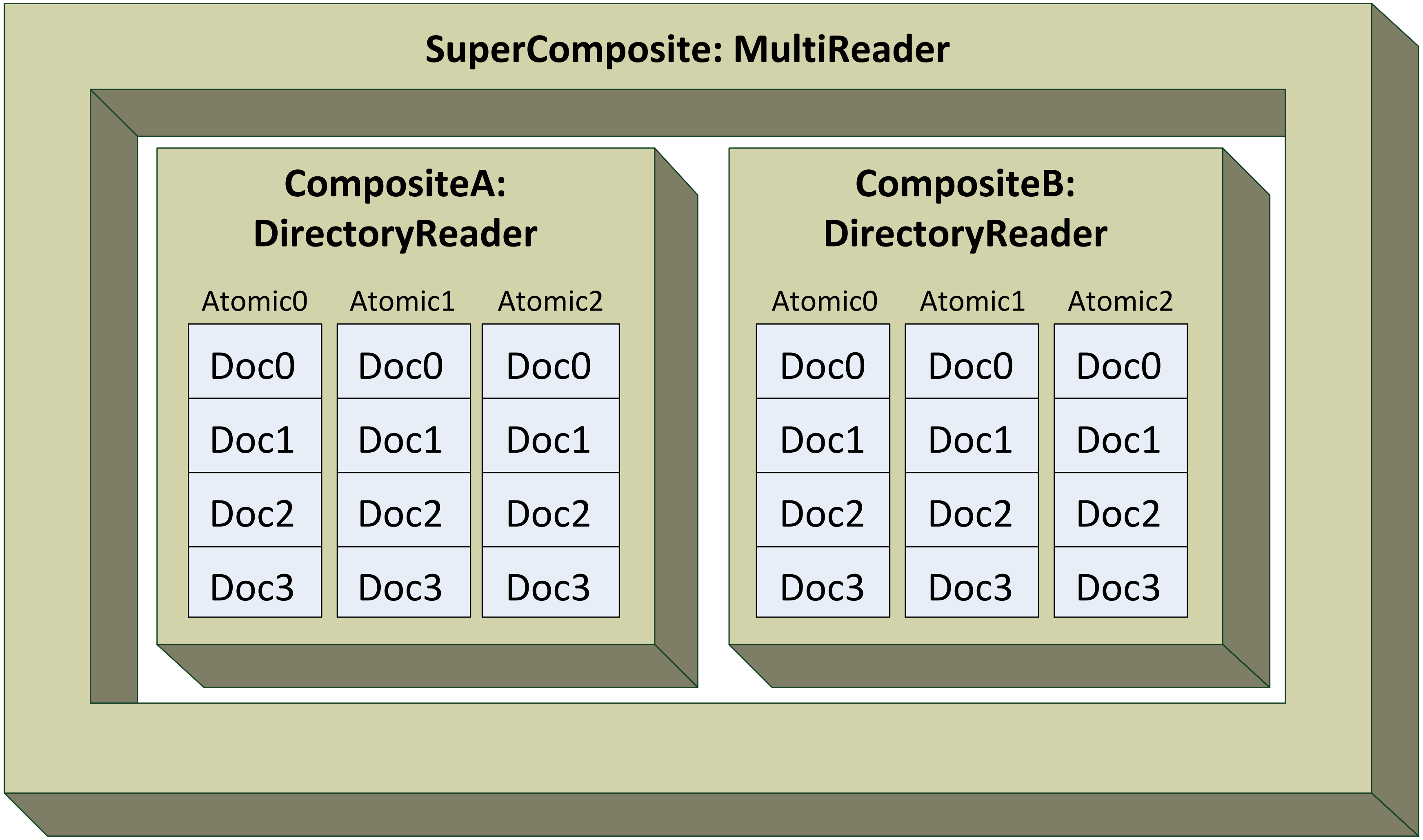
IndexReaderContext

AtomicReaderContext

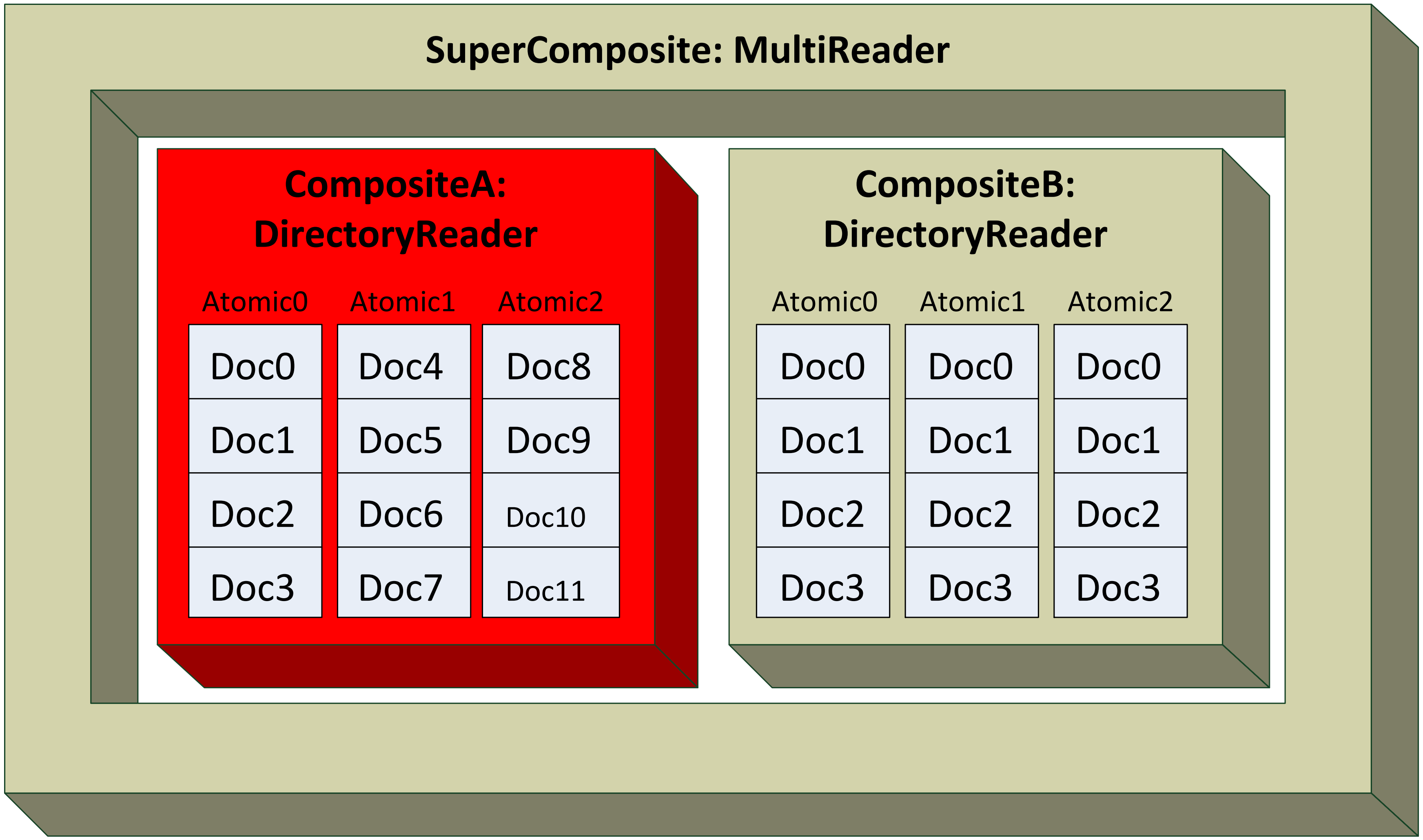
CompositeReaderContext

WTF ?!?

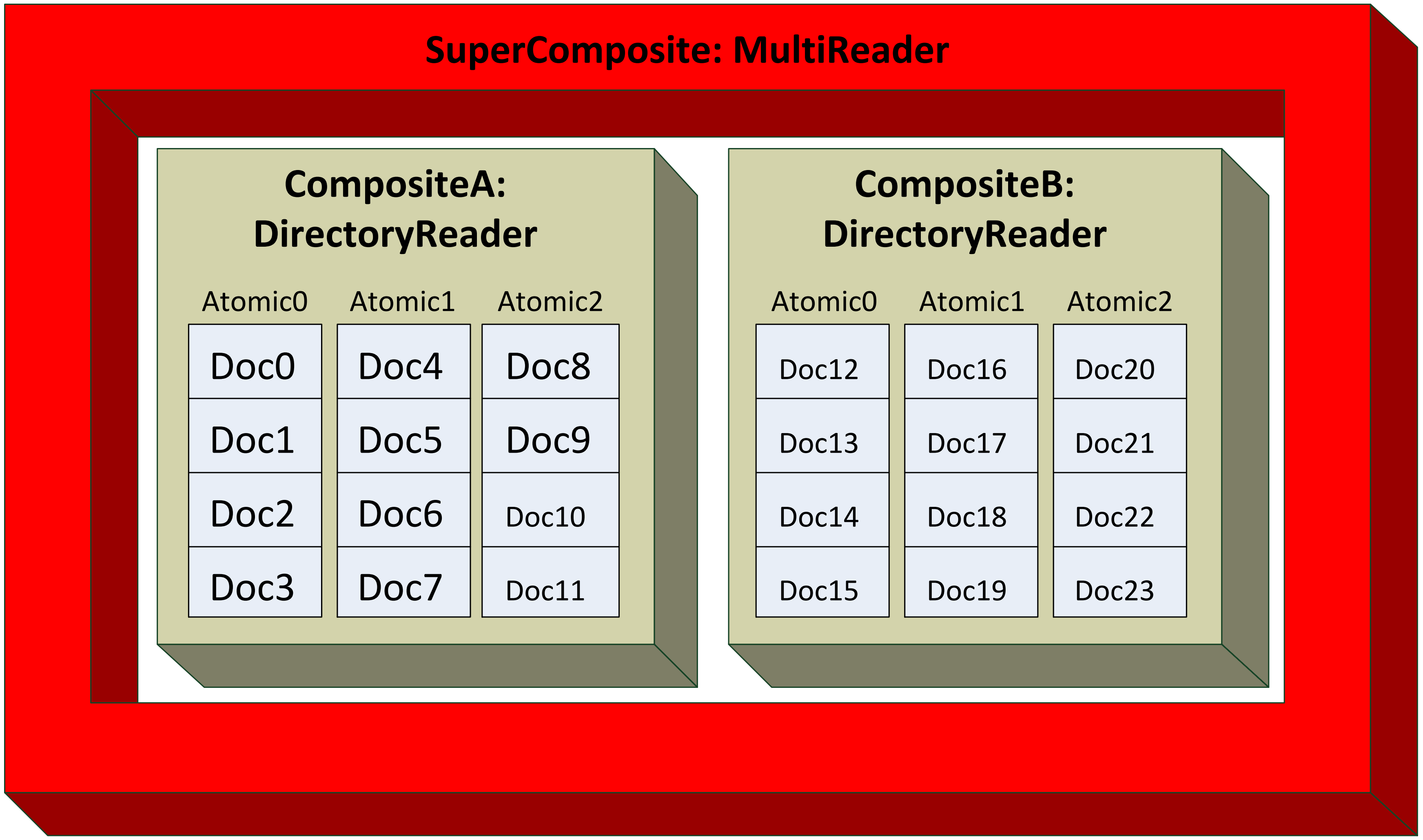
The Problem


SuperComposite: MultiReader
CompositeA: DirectoryReader
Atomic0
Atomic1
Atomic2
Doc0
Doc1
Doc2
Doc3
CompositeB: DirectoryReader
Atomic0
Atomic1
Atomic2
Doc0
Doc1
Doc2
Doc3

The Problem


SuperComposite: MultiReader
CompositeA: DirectoryReader
Atomic0
Atomic1
Atomic2
Doc0
Doc1
Doc2
Doc3
Doc4
Doc5
Doc6
Doc7
Doc8
Doc9
Doc10
Doc11
CompositeB: DirectoryReader
Atomic0
Atomic1
Atomic2
Doc0
Doc1
Doc2
Doc3
Doc0
Doc1
Doc2
Doc3
Doc0
Doc1
Doc2
Doc3

The Problem


SuperComposite: MultiReader
CompositeA: DirectoryReader
Atomic0
Atomic1
Atomic2
Doc0
Doc1
Doc2
Doc3
Doc4
Doc5
Doc6
Doc7
Doc8
Doc9
Doc10
Doc11
CompositeB: DirectoryReader
Atomic0
Atomic1
Atomic2
Doc12
Doc13
Doc14
Doc15
Doc16
Doc17
Doc18
Doc19
Doc20
Doc21
Doc22
Doc23

Solution

- Each IndexReader instance provides:
  `IndexReaderContext getTopReaderContext()`

Solution

- Each IndexReader instance provides:
  `IndexReaderContext getTopReaderContext()`
- The context provides a “view” on all childs *(direct descendants)* and leaves *(down to lowest level AtomicReaders)*

Solution

- Each IndexReader instance provides:
  `IndexReaderContext getTopReaderContext()`
- The context provides a "view" on all childs *(direct descendants)* and leaves *(down to lowest level AtomicReaders)*
- Each atomic leave has a `docBase` (document ID offset)

Solution

- Each IndexReader instance provides:
  `IndexReaderContext getTopReaderContext()`
- The context provides a “view” on all childs *(direct descendants)* and leaves *(down to lowest level AtomicReaders)*
- Each atomic leave has a `docBase` (document ID offset)
- IndexSearcher passes a context instance relative to its own top-level reader to each Query-Scorer / Filter
  - allows to access complete reader tree up to the current top-level reader
  - Allows to get the “global” document ID

Agenda

- Motivation / History of Lucene
- AtomicReader & CompositeReader
- Reader contexts
- Wrap up

Summary

- Lucene moved from global searches to **per-segment search** in *Lucene 2.9*
- Up to *Lucene 3.6* indexes are still accessible on any hierarchy level **with same interface**
- *Lucene 4.0* will **split the IndexReader class** into several abstract interfaces
- IndexReaderContexts will support **per-segment search preserving top-level document IDs**

Contact

Uwe Schindler

uschindler@apache.org
http://www.thetaphi.de
@thetaph1

SD DataSolutions GmbH
Wätjenstr. 49
28213 Bremen, Germany
+49 421 40889785-0
http://www.sd-datasolutions.de