Realtime Search with Lucene

Michael Busch
@michibusch
michael@twitter.com
buschmi@apache.org

Realtime Search with Lucene

Agenda

- Introduction
  - Near-realtime Search (NRT)
  - Searching DocumentsWriter's RAM buffer
  - Sequence IDs
  - Twitter prototype
  - Roadmap

Introduction

Introduction

- Lucene made great progress towards realtime search with the Near-realtime search feature (NRT) added in 2.9

- NRT reduces search latency (time it takes until a document becomes searchable) significantly, using the new `IndexWriter.getReader()`

- Drawback of NRT: If `getReader()` is called frequently, indexing performance decreases significantly

- New approach: Searching on IndexWriter's/DocumentsWriter's in-memory buffer directly

Realtime Search with Lucene

Agenda

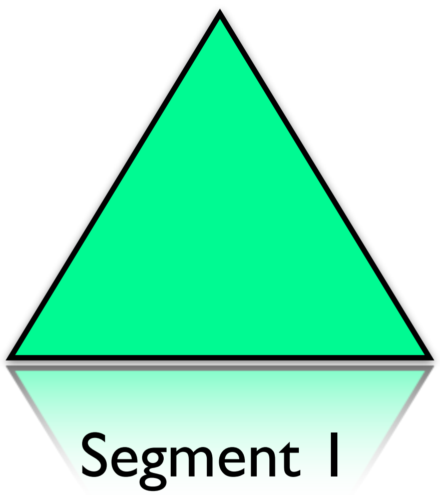
- Introduction
- Near-realtime Search (NRT)
- Searching DocumentsWriter’s RAM buffer
- Sequence IDs
- Twitter prototype
- Roadmap

Near-realtime Search (NRT)

Incremental Indexing

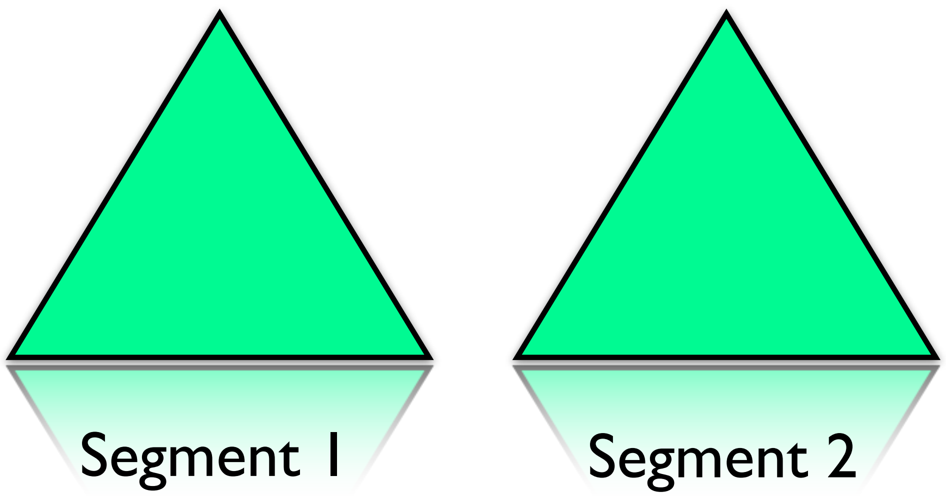
- Lucene is an incremental indexer - documents can be added to an existing, searchable index
- Lucene writes “segments”, which are small indexes itself
- A Lucene index consists of one or more segments
- Small segments are merged into larger ones to limit total number of segments per index

Incremental Indexing



- After a segment is written and committed (triggered by **`IndexWriter.commit()`** or **`IndexWriter.close()`**) it is visible to IndexReaders

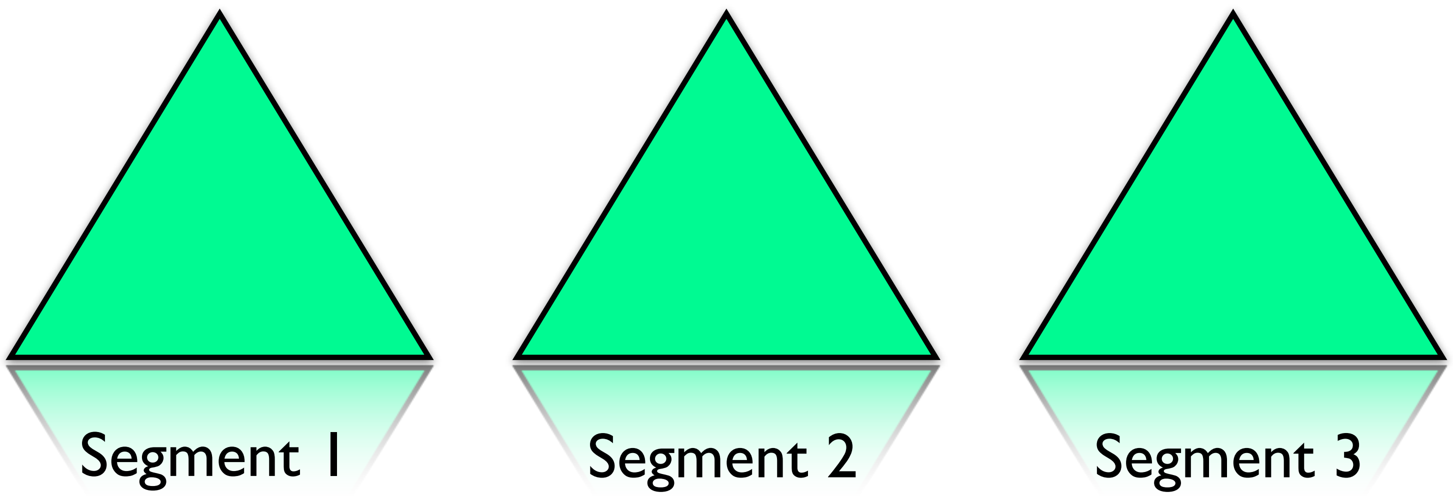
Incremental Indexing



- After a segment is written and committed (triggered by `IndexWriter.commit()` or `IndexWriter.close()`) it is visible to IndexReaders

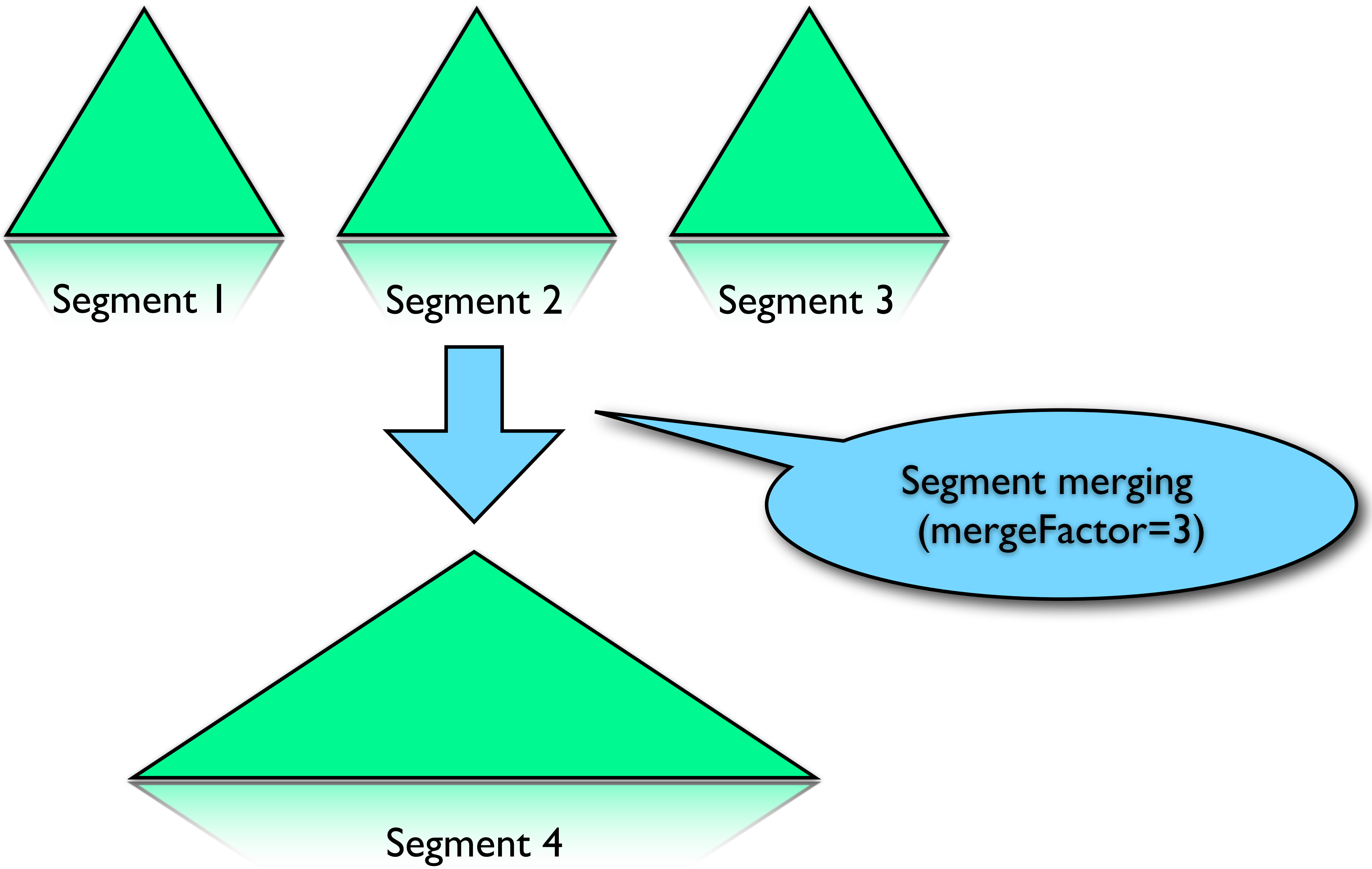
- New segments can be written, while IndexReaders execute queries on older segments

Incremental Indexing

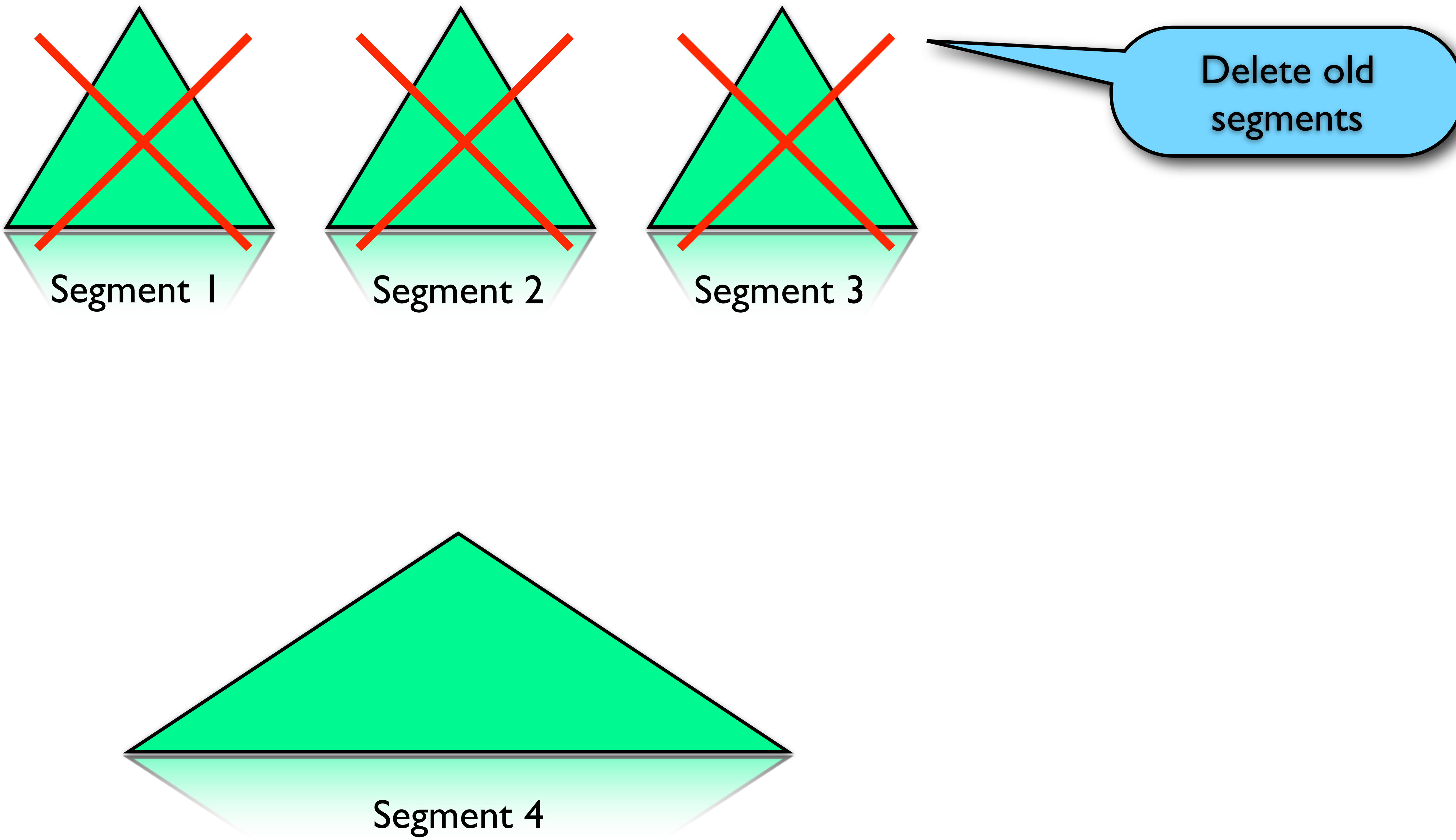


- After a segment is written and committed (triggered by **`IndexWriter.commit()`** or **`IndexWriter.close()`**) it is visible to IndexReaders
- New segments can be written, while IndexReaders execute queries on older segments

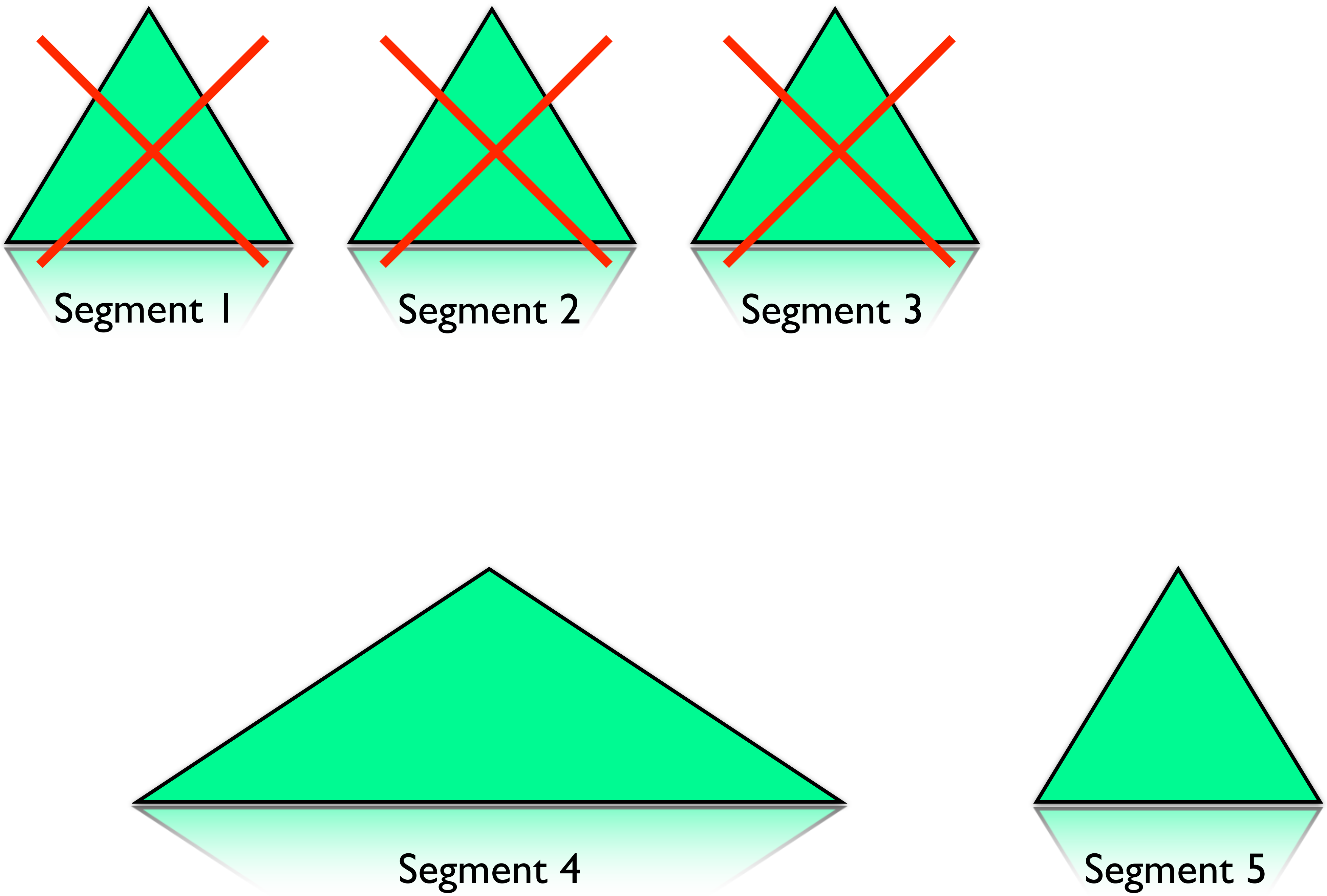
Incremental Indexing



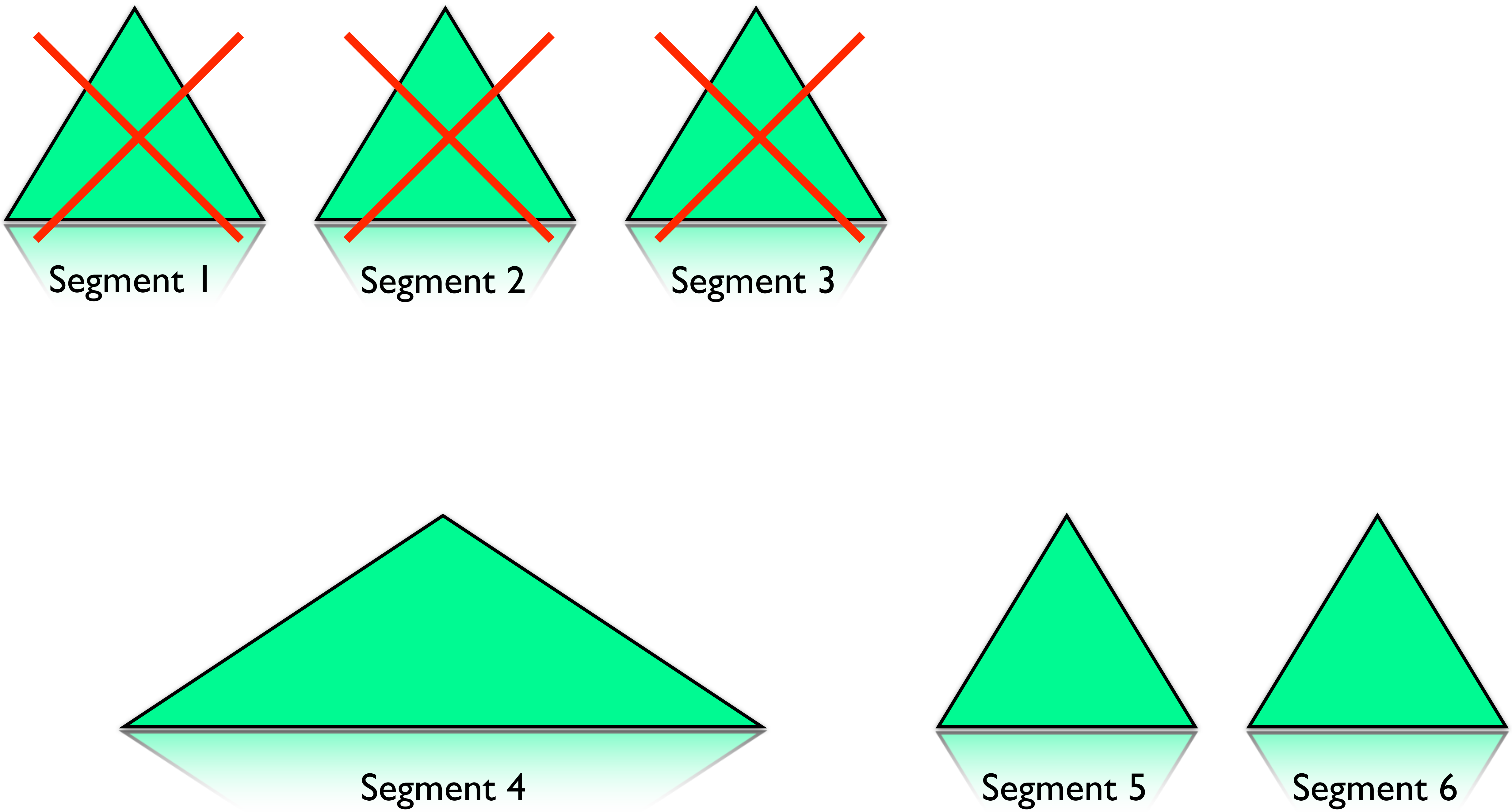
Incremental Indexing



Incremental Indexing



Incremental Indexing



Committing an index segment

- Flush in-memory data structures to index location (usually on disk)
- Possibly trigger a segment merge
- Synchronize segment files, which forces the OS to flush those files from the FS cache to the physical disk (this can be an expensive operation)
- Append an entry to segments_x file and write new segment_x+1 file
- `IndexWriter.close()` might have to wait for in-flight segment merges to complete (this can be **very** expensive)

Near-realtime search (NRT)

- NRT tries to avoid the two most expensive aspects of segment committing: file handle sync calls and waiting for segment merge completion

- **`IndexWriter.getReader()`** can be called to obtain an IndexReader, that can query **flushed, not-yet-committed** segments

- Reduces indexing latency significantly, and IndexWriters don't have to be closed to (re)open IndexReaders

- Disadvantage: **`getReader()`** triggers a flush of the in-memory data structures

A little bit Lucene history: LUCENE-843

- Indexer was rewritten with LUCENE-843 patch (released in 2.3)
- Indexing performance improved by 5x-10x (!!)
- Before, each document was inverted and encoded as its own segment
- These tiny single-doc segments were merged with Lucene's standard SegmentMerger
- LUCENE-843 introduced class DocumentsWriter, which can take a large number of docs and invert them into a single segment
- Dramatic improvements in memory consumption and indexing performance

Near-realtime search (NRT)

- IndexWriter.getReader() triggers DocumentsWriter to flush its in-memory data structures into a segment every time it’s called

- If called very frequently (desired in realtime search), it results in a similar behavior as before LUCENE-843

Realtime Search with Lucene

Agenda

- Introduction
- Near-realtime Search (NRT)
- Searching DocumentsWriter's RAM buffer
- Sequence IDs
- Twitter prototype
- Roadmap

Searching DocumentsWriter's RAM buffer

Goals

- **Goal 1:**
  Allow IndexReaders to search on DocumentsWriter's RAM buffer, while documents are being appended simultaneously to the same data structures

- **Goal 2:**
  Maintain high indexing performance with large RAM buffer, and independent of the query load

- **Goal 3:**
  Opening a RAM IndexReader should be so cheap, so that a new reader can be opened for every query (drops latency close to zero)

LUCENE-2329: Parallel posting arrays

- Already committed to Lucene's trunk
- Changes how per-term data is stored in RAM

Inverted Index

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

Example from:
Justin Zobel , Alistair Moffat,
Inverted files for text search engines,
ACM Computing Surveys (CSUR)
v.38 n.2, p.6-es, 2006

Inverted Index

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

term	freq	
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>

Dictionary and posting lists

Inverted Index

Query: keeper

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

term	freq	
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>

Dictionary and posting lists

Inverted Index

Query: keeper

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

term	freq	
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>

Dictionary and posting lists

Inverted Index

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

term	freq	
and		<6>
big		
dark		
did		
gown		
had		
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>

Per term we store different kinds of metadata: text pointer, frequency, postings pointer, etc.

Dictionary and posting lists

LUCENE-2329: Parallel posting arrays

```
class PostingList

int textPointer;
int postingsPointer;
int frequency;
...
```

- Term hashtable is an array of these objects: **`PostingList[] termsHash`**
- For each unique term in a segment we need an instance; this results in a very large number of objects that are long-living, i.e. the garbage collecter can't remove them quickly (they need to stay in memory until the segment is flushed)
- With a searchable RAM buffer we want to flush much less often and allow DocumentsWriter to fill up the available memory

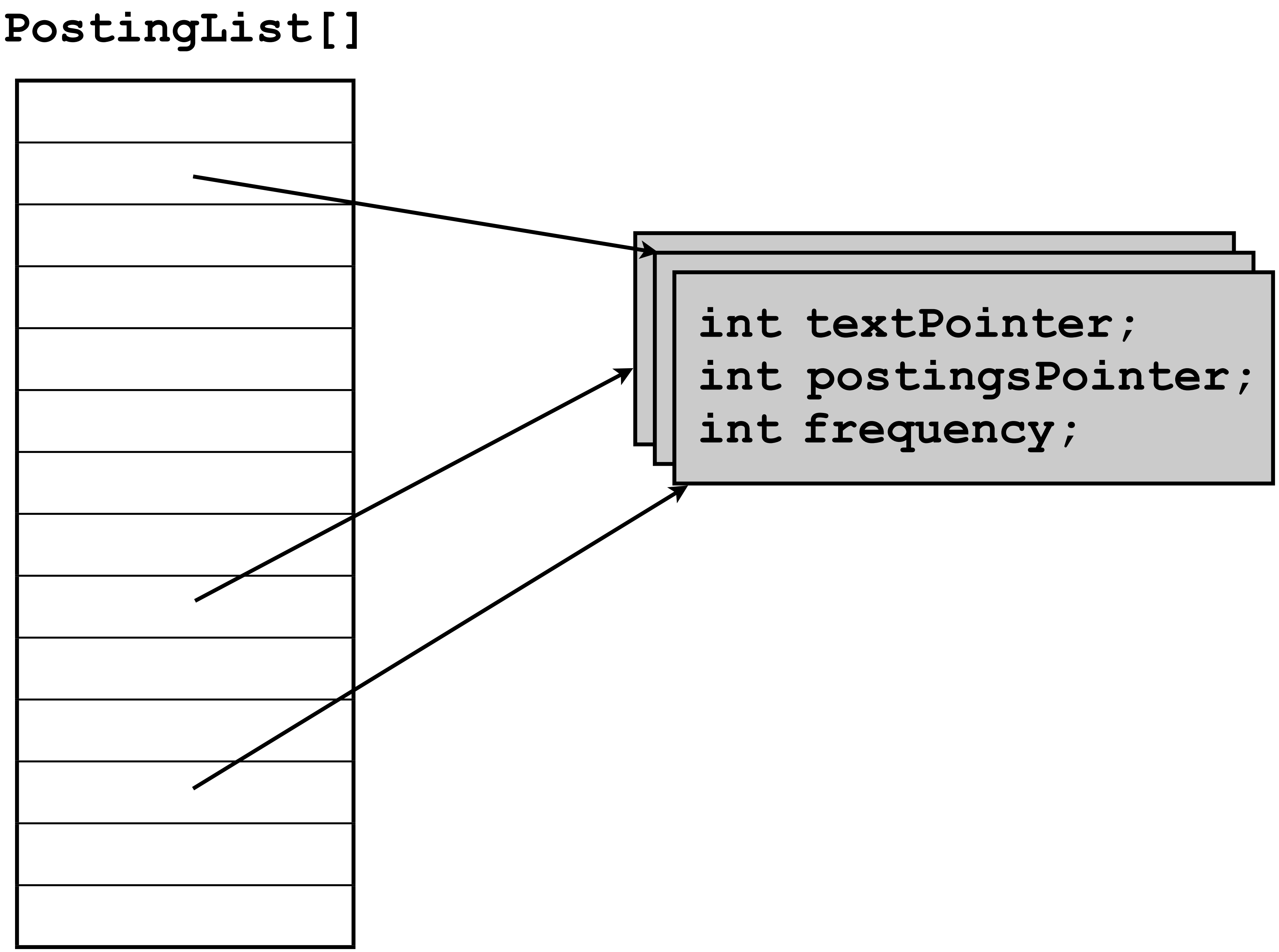
LUCENE-2329: Parallel posting arrays

```
class PostingList
```

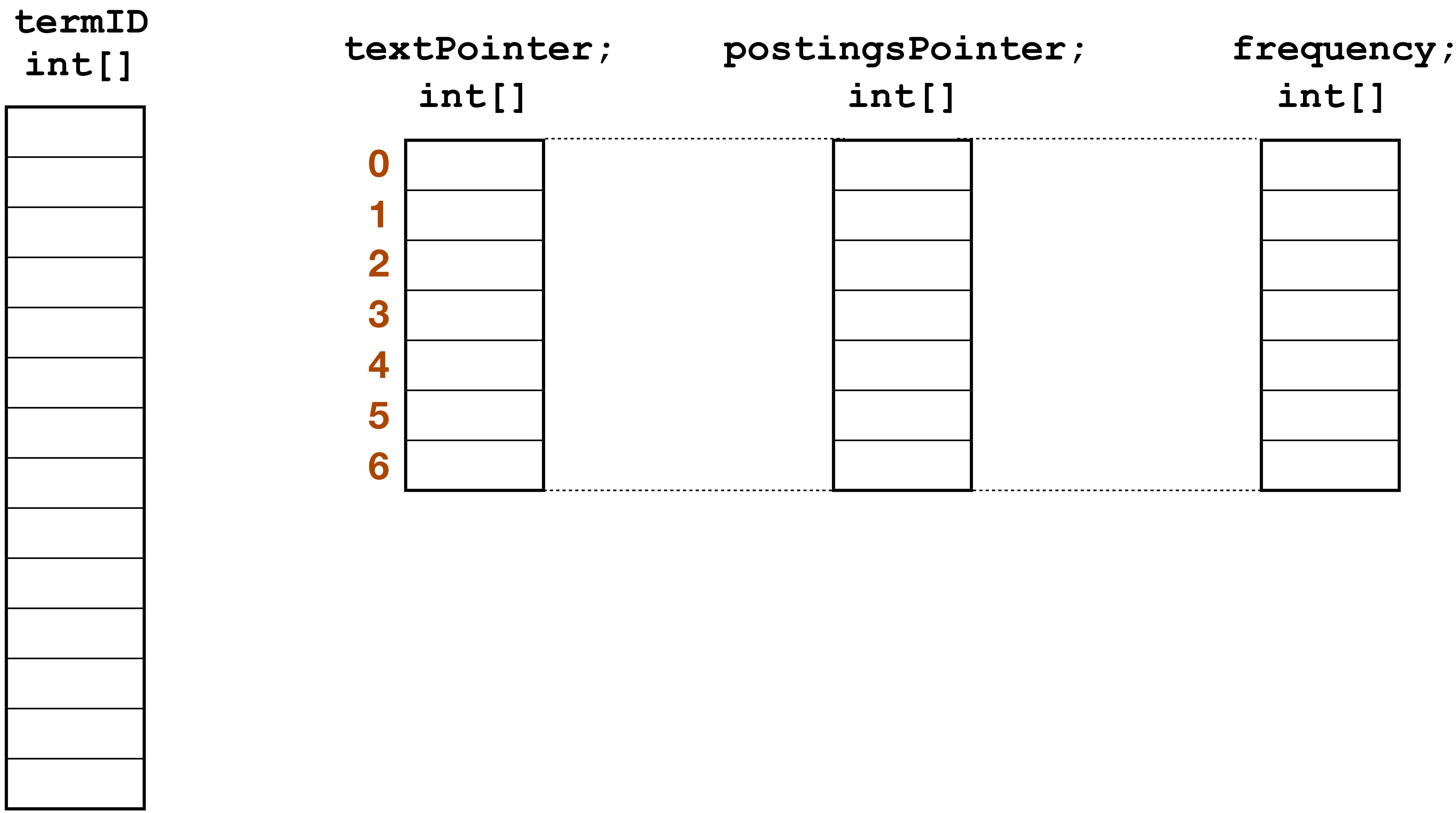
```
int textPointer;
int postingsPointer;
int frequency;
...
```

- Having a large number of long-living objects is very expensive in Java, especially when the default mark-and-sweep garbage collector is used
- The mark phase of GC becomes very expensive, because all long-living objects in memory have to be checked
- We need to reduce the number of objects to improve GC performance!
  -> Parallel posting arrays

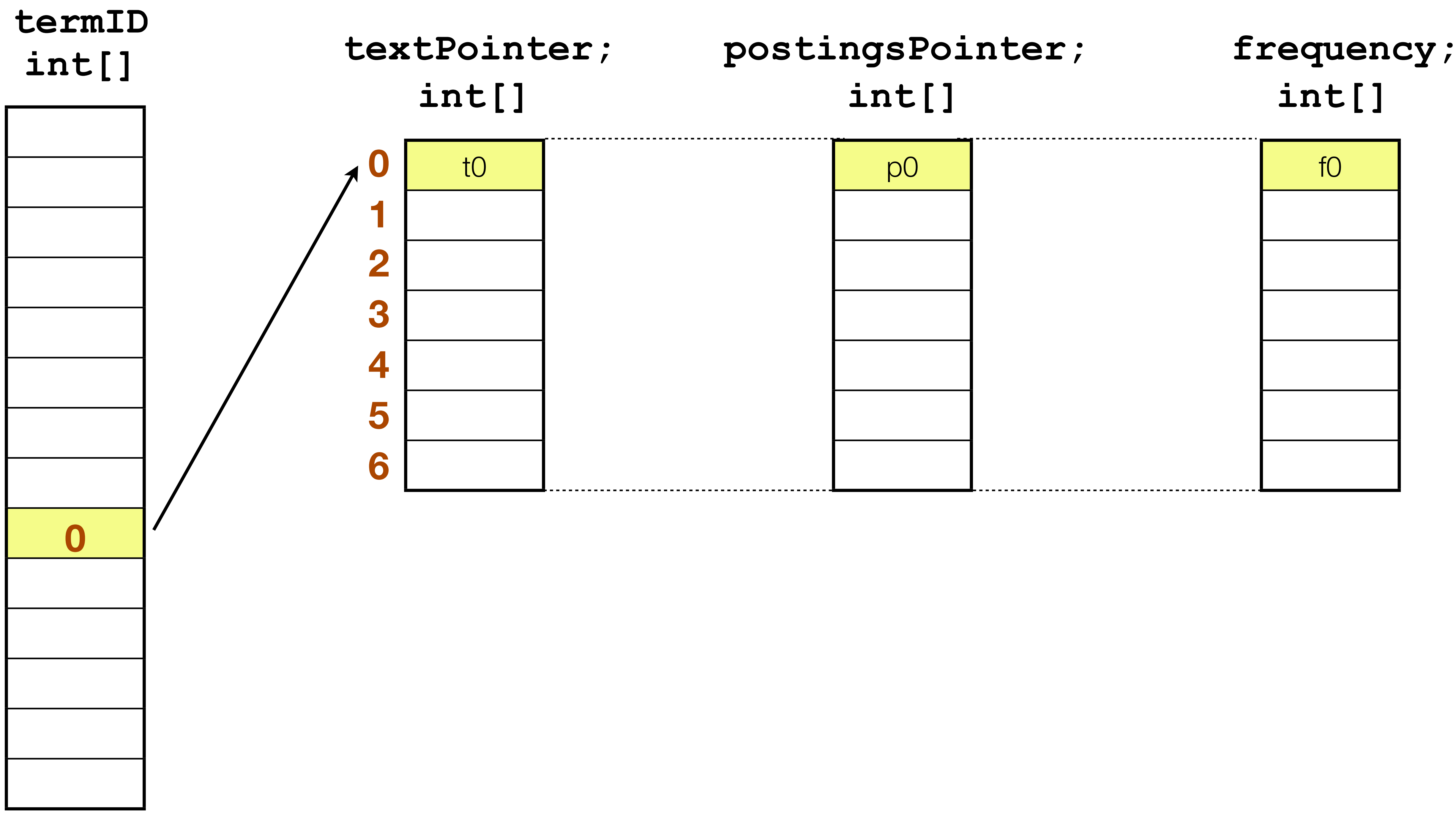
LUCENE-2329: Parallel posting arrays



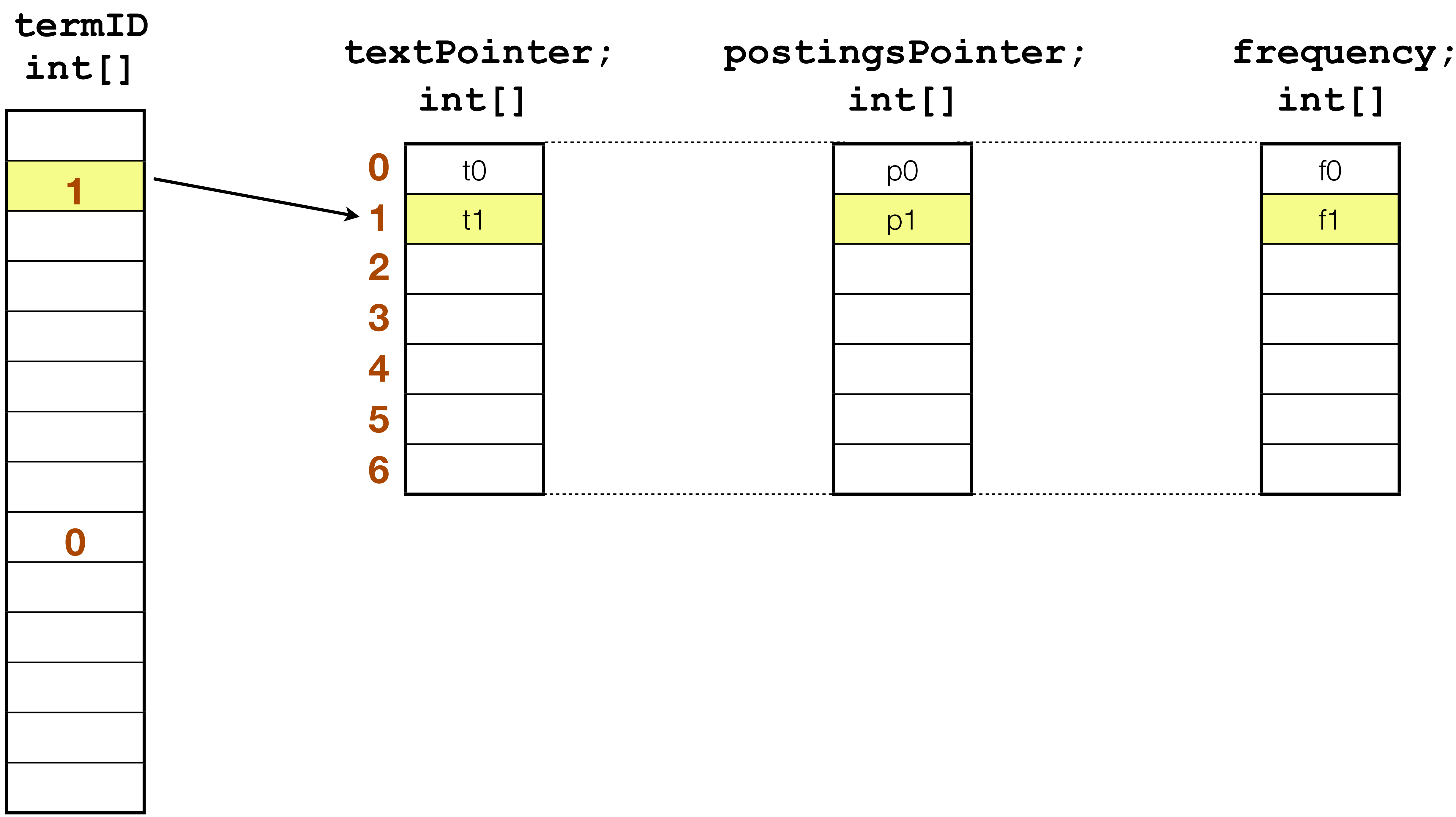
LUCENE-2329: Parallel posting arrays



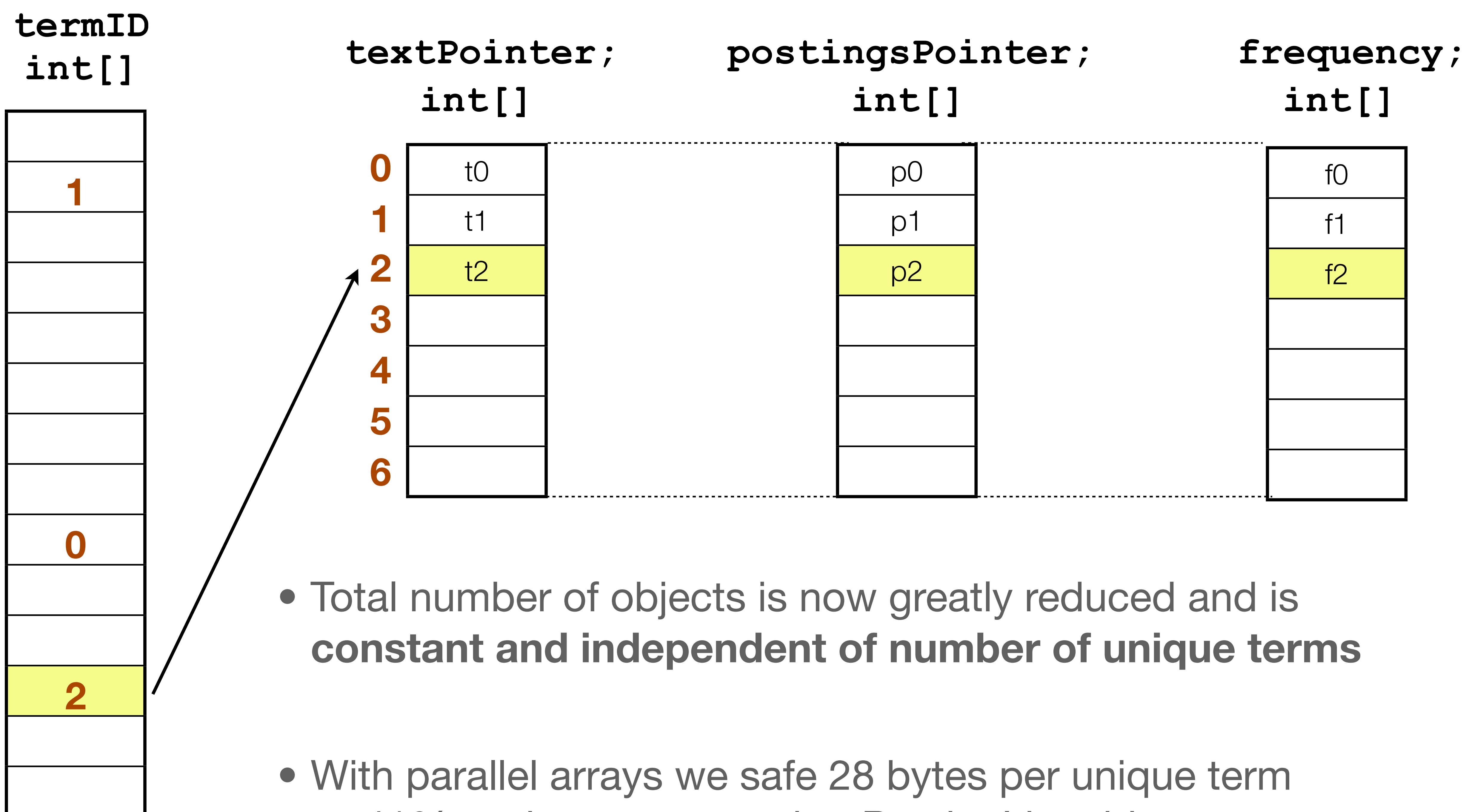
LUCENE-2329: Parallel posting arrays



LUCENE-2329: Parallel posting arrays



LUCENE-2329: Parallel posting arrays



- Total number of objects is now greatly reduced and is **constant and independent of number of unique terms**
- With parallel arrays we safe 28 bytes per unique term -> 41% savings compared to PostingList object

LUCENE-2329: Parallel posting arrays - Performance

- Performance experiments: Index 1M wikipedia docs

1) -Xmx**2048M**, indexWriter.setMaxBufferSizeMB(**200**)

4.3% improvement

2) -Xmx**256M**, indexWriter.setMaxBufferSizeMB(**200**)

86.5% improvement

LUCENE-2329: Parallel posting arrays - Performance

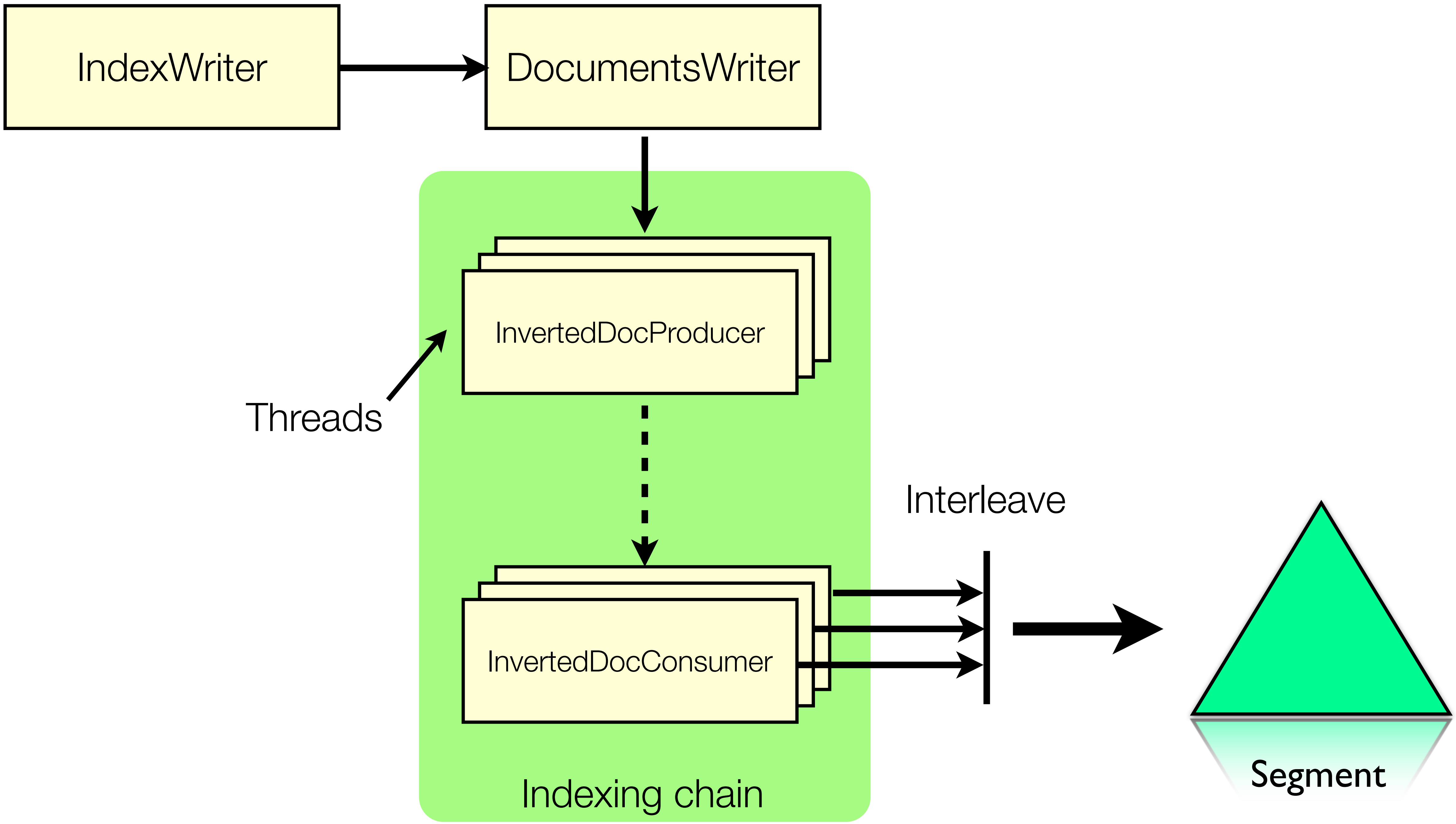
- With large heap there is a small improvement due to per-term memory savings

- With small heap the garbage collector is invoked much more often - huge improvement due to smaller number of objects (depending on doc sizes we have seen improvements of up to **400%**!)

- With searchable RAM buffers we want to utilize all the RAM we have; with parallel arrays we can maintain high indexing performance even if we get close to the max heap size

Goal 2:
Maintain high indexing performance with large RAM buffer, and independent of the query load

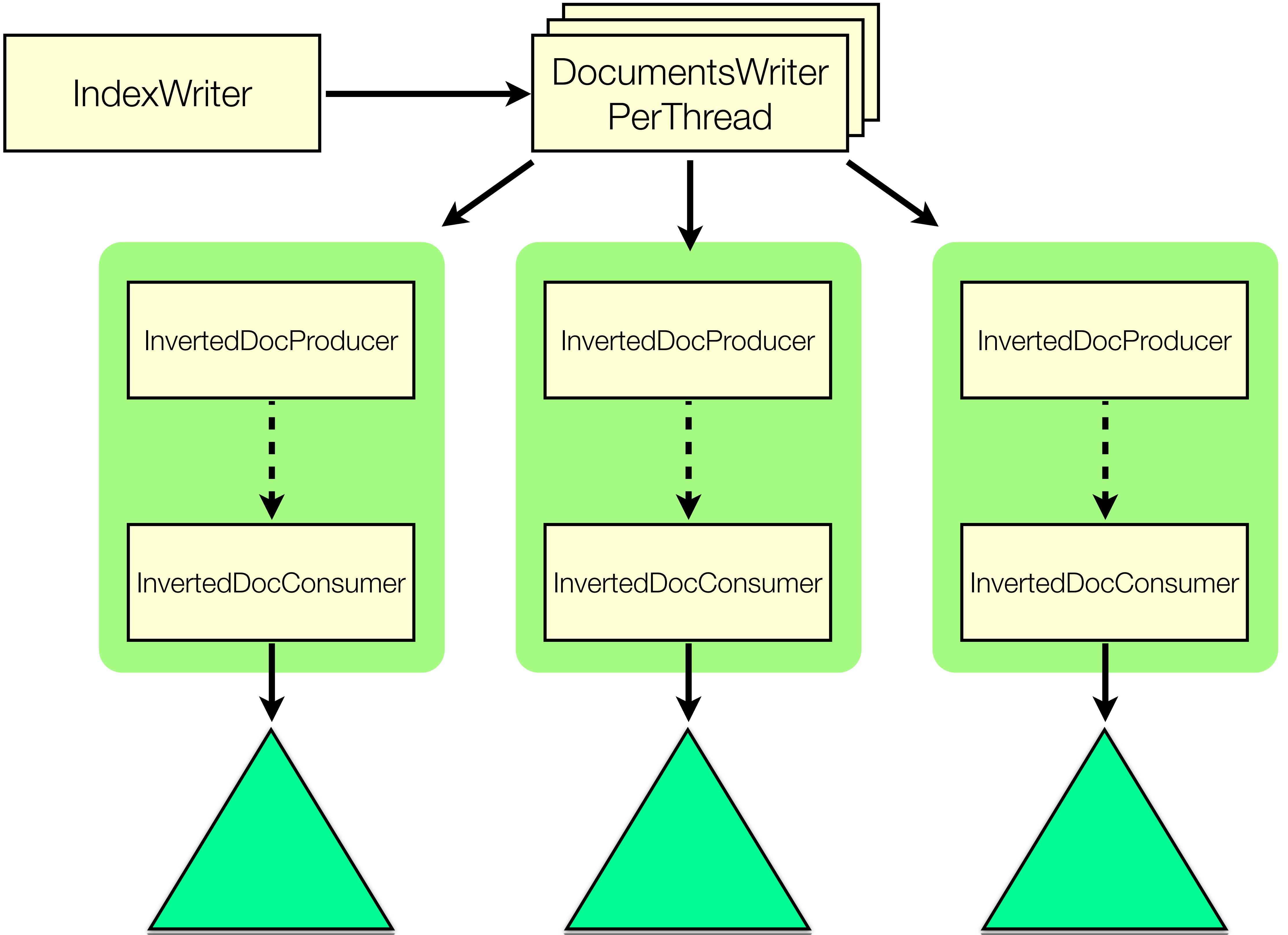
Today: Multi-threaded Indexing chain



Today: Multi-threaded Indexing chain

- The interleaving step is quite expensive
- Flushing “stops the world”: No documents can be added during flushing/ interleaving
- Multi-threaded code necessary in all IndexingChain classes, e.g. we have >10 *PerThread classes in the indexer package

LUCENE-2324: Single threaded indexing chain



LUCENE-2324: Single threaded indexing chain

- Multiple per-thread DocumentsWriters write their own private segments
- Great simplification, many perThread classes can be removed (see 2324 patch)
- DocumentsWriterPerThreads can flush independently without “stopping the world”; interleaving step not necessary anymore
- This change reduces the concurrency problem we need to solve for RAM IndexReaders to a single-writer, multi-reader problem -> lock-free algorithms are now possible

Searching DocumentsWriter's RAM buffer

- Implement an IndexReader that shares the indexes data structures with DocumentsWriter
- Terms hashtable is used for fast term lookup
- TermDocs/TermPositions implementation for in-memory postinglists
- Sequence IDs for efficient deletes
- IndexReader needs to be able to switch automatically and on-the-fly from reader the RAM buffer to a flushed segment in case DocumentsWriter flushes its buffer while searches are in-flight

Goal 1:
Allow IndexReaders to search on DocumentsWriter's RAM buffer, while documents are being appended simultaneously to the same data structures

Concurrency

- Having a single writer thread simplifies our problem: no locks have to be used to protect data structures from corruption (only one thread modifies data)

- But: we have to make sure that all readers **always** see a consistent state of **all** data structures -> this is much harder than it sounds!

- In Java, it is not guaranteed that one thread will see changes that another thread makes in program execution order, unless the same memory barrier is crossed by both threads -> **safe publication**

- Safe publication can be achieved in different, subtle ways. Read the great book “Java concurrency in practice” by Brian Goetz for more information!

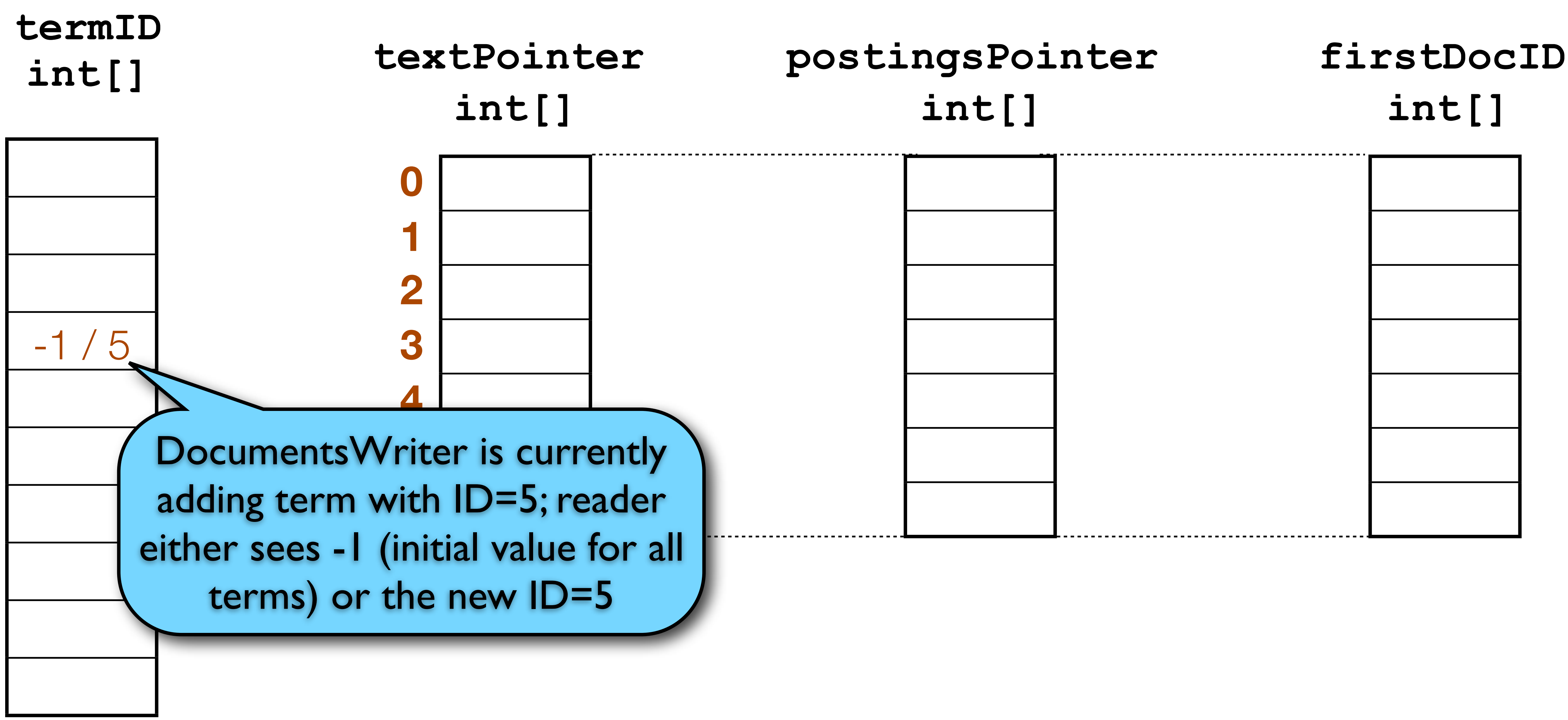
- Going through all details could easily fill an entire talk. We’ll only look into a few examples here.

Concurrency - Example: term lookup

- Each reader remembers the max. docID of the last completely indexed document at the time the reader was opened
- For each term we store the first docIDs it occurred in. We make sure the parallel array holding those first docIDs is properly initialized (visible to readers)
- When we lookup a term with an IndexReader, we compare the reader's maxDocID with the first docID of the term; the term is only returned if maxDocID(reader) >= firstDocID(term); otherwise the lookup method returns term_not_found
- There are not "dirty reads" on integers in Java, meaning a thread either gets the old or the new value of a variable that another thread is writing too in parallel

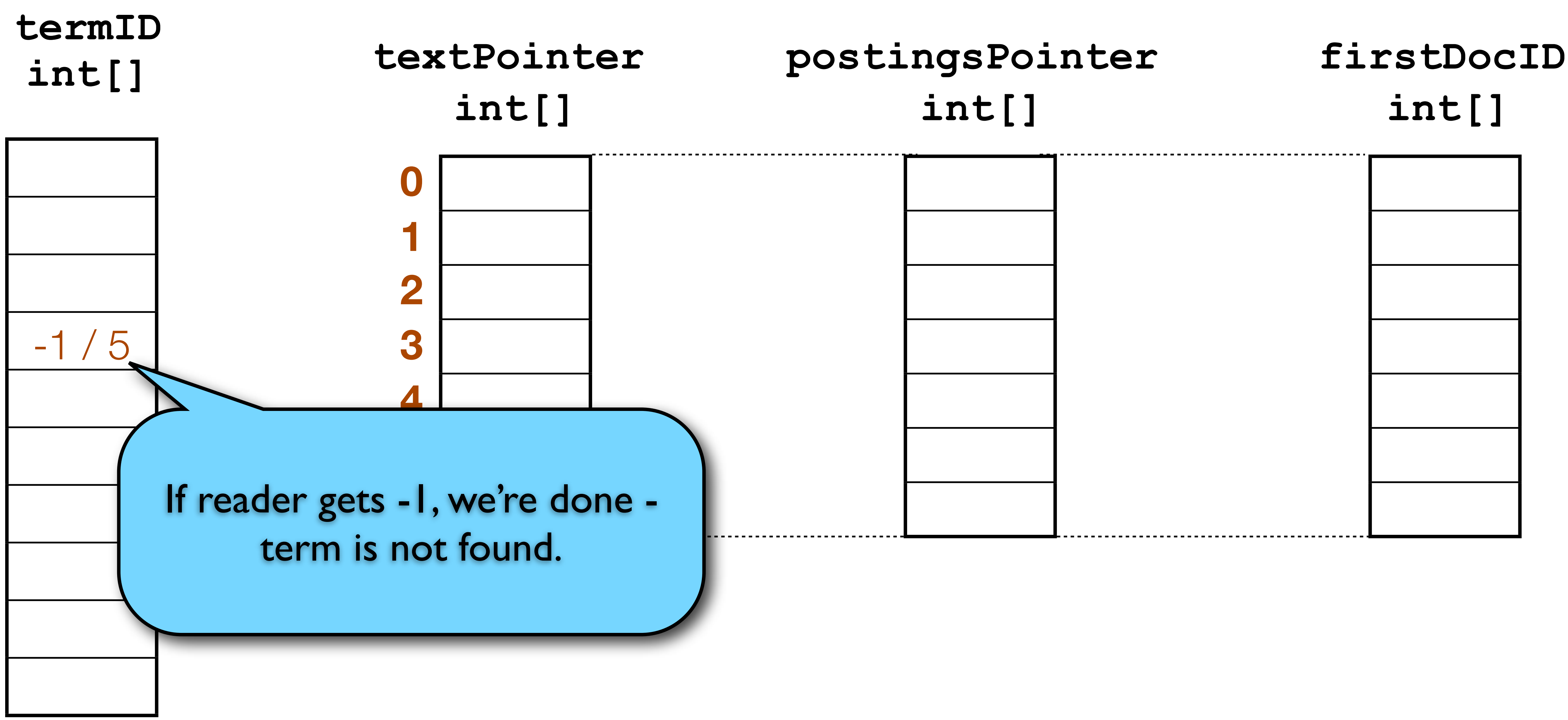
Concurrency - Example: term lookup

- If a reader tries to lookup a term that a writer is at the same time writing for the first time (term has not yet occurred in earlier documents) different things can happen:



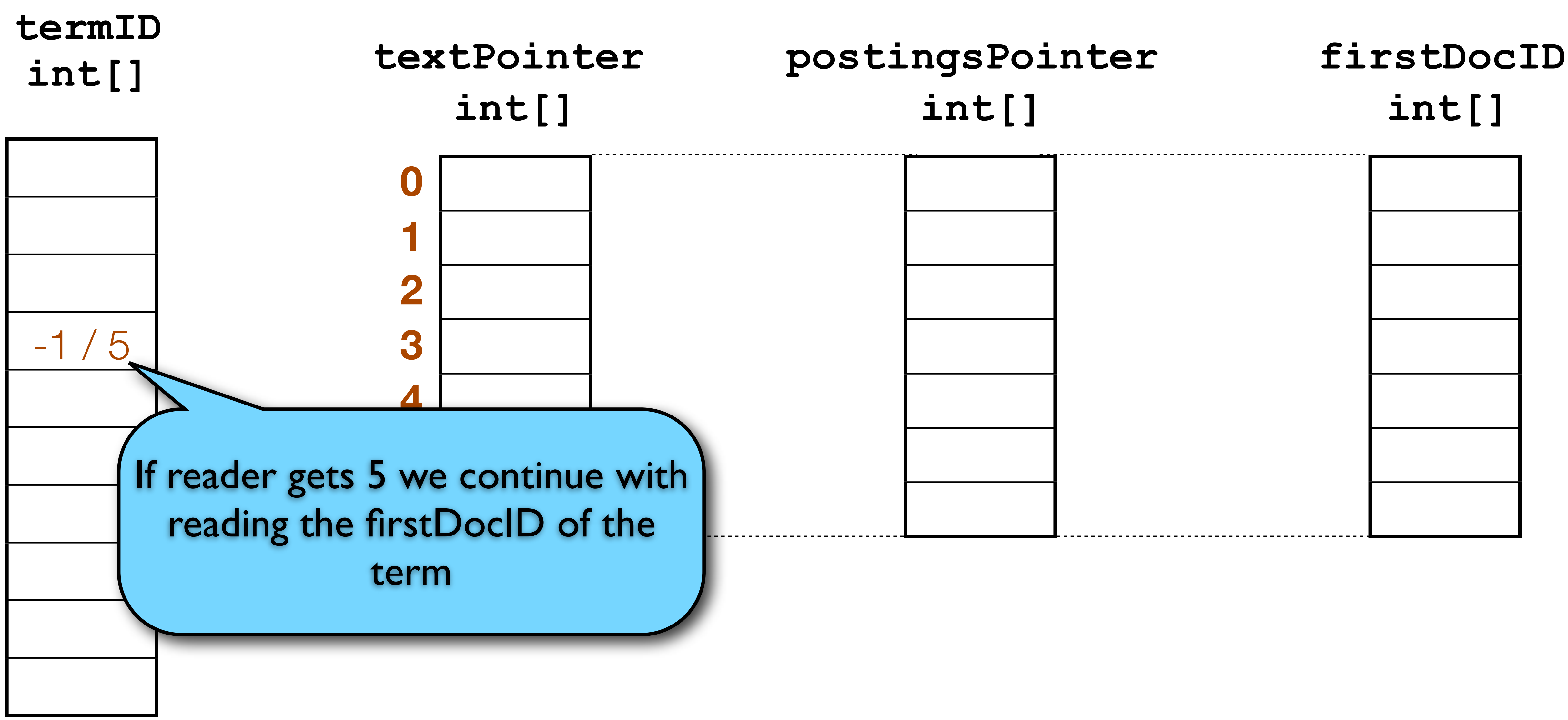
Concurrency - Example: term lookup

- If a reader tries to lookup a term that a writer is at the same time writing for the first time (term has not yet occurred in earlier documents) different things can happen:



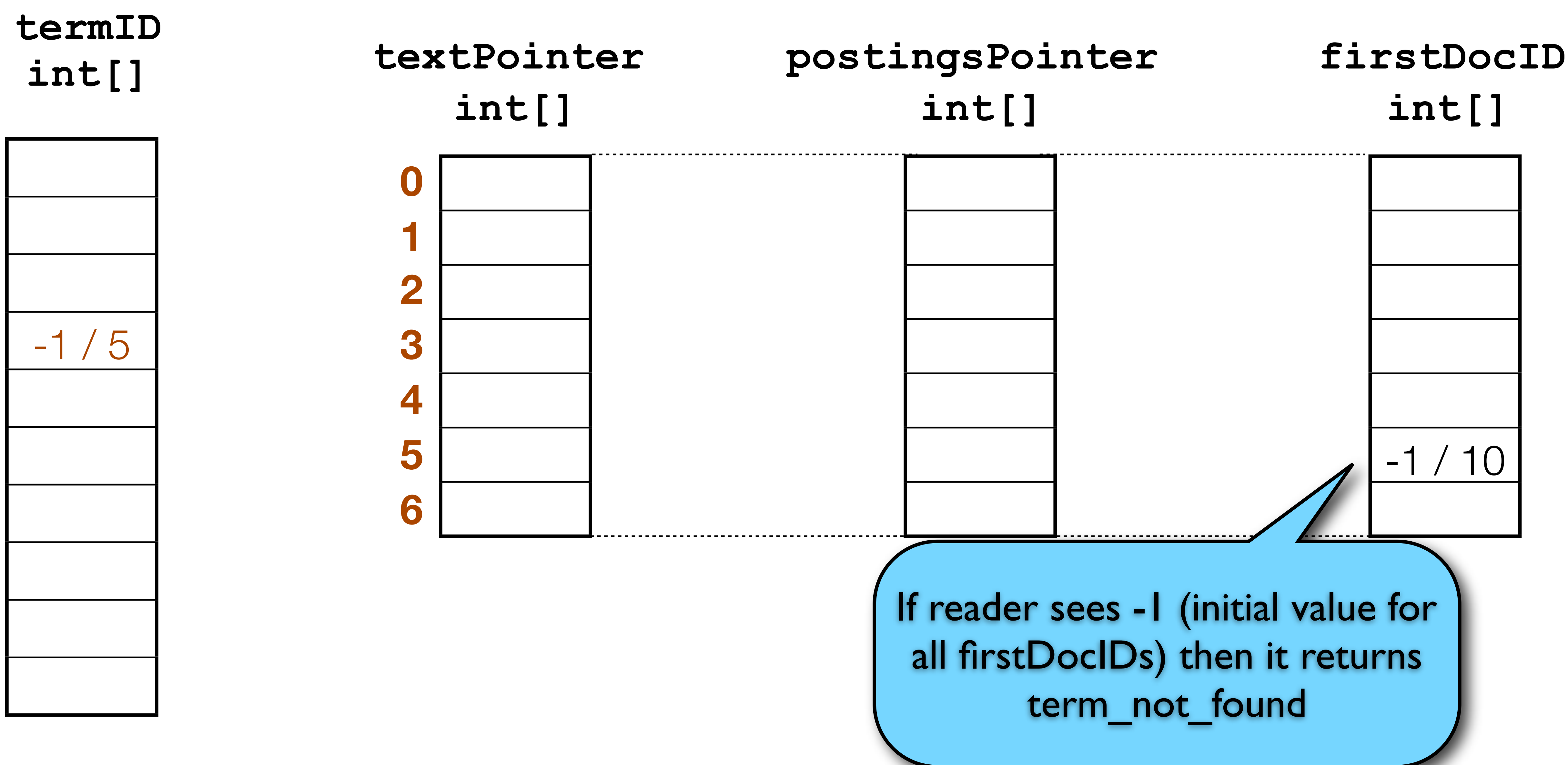
Concurrency - Example: term lookup

- If a reader tries to lookup a term that a writer is at the same time writing for the first time (term has not yet occurred in earlier documents) different things can happen:



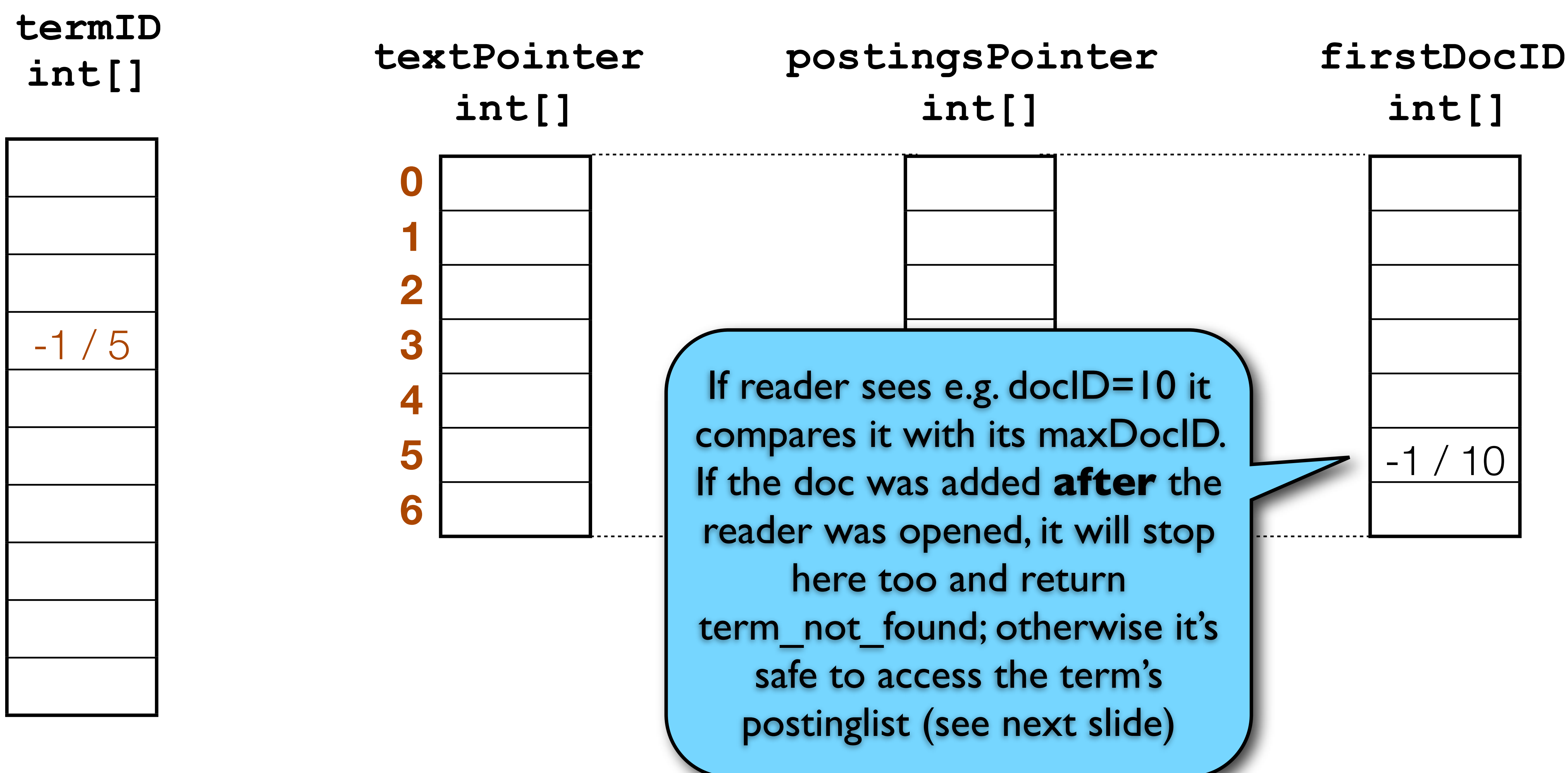
Concurrency - Example: term lookup

- If a reader tries to lookup a term that a writer is at the same time writing for the first time (term has not yet occurred in earlier documents) different things can happen:



Concurrency - Example: term lookup

- If a reader tries to lookup a term that a writer is at the same time writing for the first time (term has not yet occurred in earlier documents) different things can happen:



Concurrency - Example: term lookup

- After each document is fully indexed the writer thread is forced to cross a memory barrier

- When a reader is opened the opening thread is also forced to cross the same memory barrier

- A memory barrier can be as simple as a single volatile variable that multiple threads access

- Hence, visibility for all documents older than maxDocID is ensured for an IndexReader

Realtime Search with Lucene

Agenda

- Introduction
- Near-realtime Search (NRT)
- Searching DocumentsWriter's RAM buffer
- ‣ Sequence IDs
- Twitter prototype
- Roadmap

Sequence IDs

IndexWriter API

- `void addDocument(Document doc);`
- `void updateDocument(Term delTerm, Document doc);`
- `void deleteDocuments(Term delTerm);`
- `void commit();`
- All these methods are thread-safe
- But: in which order are they executed?

IndexWriter API - Example

term occurs in both docs

Thread 1:

```
addDoc(doc1);
addDoc(doc2);
```

Thread 2:

```
deleteDocs(term);
```

Thread 3:

```
IW.getReader();
```

- Problem: Will Thread 2 only delete doc1 or also doc2? Which state will the reader that Thread 3 opens “see”?
- Answer: It depends on Java’s thread scheduling which thread acquires the mutex first.
- It’s currently hard to write code that can track the order of calls and answer the question above.

IndexWriter API

- `long addDocument(Document doc);`
- `long updateDocument(Term delTerm, Document doc);`
- `long deleteDocuments(Term delTerm);`
- `long commit();`
- All methods will return a sequence ID, which unambiguously indicate the order the operations were executed in
- An RAM IndexReader will also have a sequence ID that defines which snapshot of the index it can “see”

IndexWriter API - Example

Thread 1:

```
addDoc(doc1); 1
addDoc(doc2); 3
```

Thread 2:

```
deleteDocs(term); 2
```

Thread 3:

```
IW.getReader(); 1
```

- doc1 is added before delete; delete happens before doc 2 is added
- Thread 3's reader will see doc 1

IndexWriter API - Example

Thread 1:

```
addDoc(doc1);   1
addDoc(doc2);   3
```

Thread 2:

```
deleteDocs(term);   2
```

Thread 3:

```
IW.getReader();   3
```

- doc1 is added before delete; delete happens before doc 2 is added
- Thread 3’s reader will only see doc 2 (doc 1 will appear as deleted)

IndexWriter API - Example

Thread 1:

```
addDoc(doc1);  1
addDoc(doc2);  2
```

Thread 2:

```
deleteDocs(term);  3
```

Thread 3:

```
IW.getReader();  2
```

- doc1 is added before doc2; delete happens after both docs are added
- Thread 3's reader will see both docs

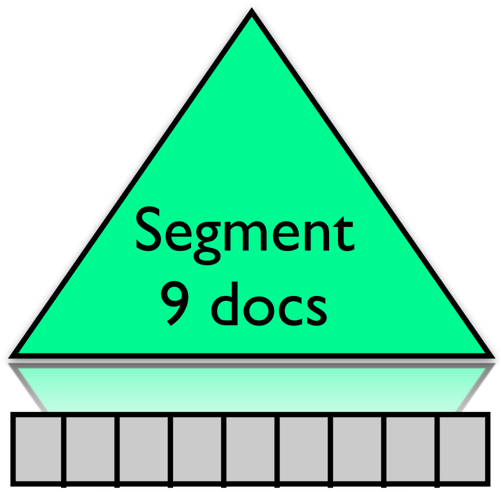
IndexWriter API - Example

Thread 1:	Thread 2:	Thread 3:
`addDoc(doc1);` 1 `addDoc(doc2);` 2	`deleteDocs(term);` 3	`IW.getReader();` 3

- doc1 is added before doc2; delete happens after both docs are added
- Thread 3's reader will not see any docs (both will appear as deleted)

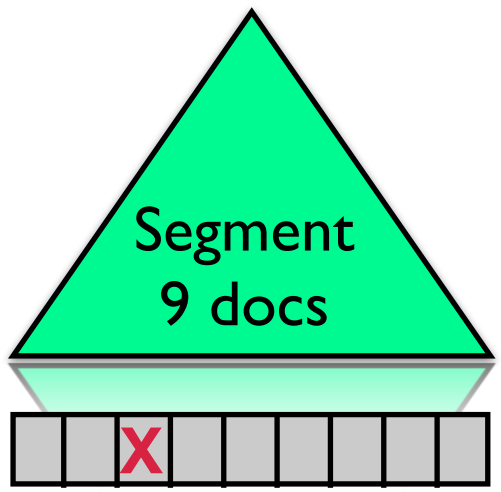
Deletes

- Today deletes are stored as BitSets



Deletes

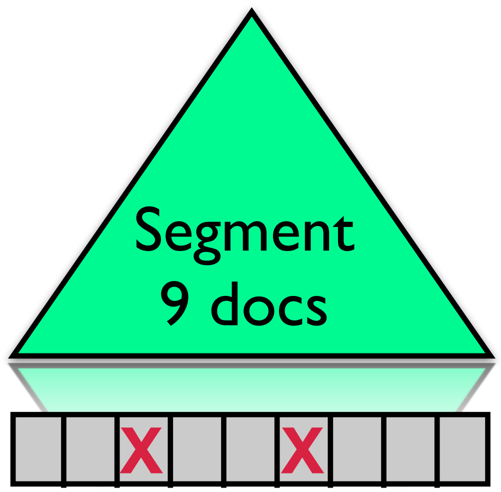
- Today deletes are stored as BitSets




```
deleteDoc(2);
```

Deletes

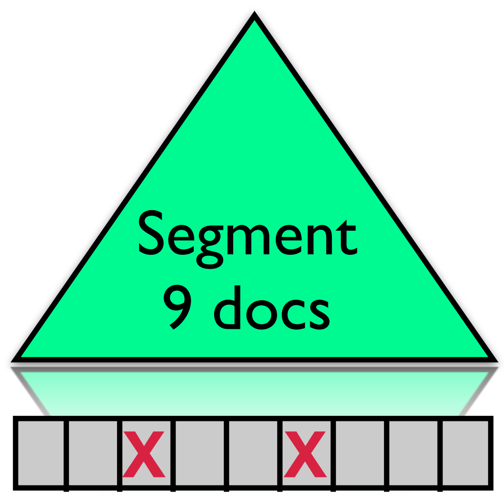
- Today deletes are stored as BitSets



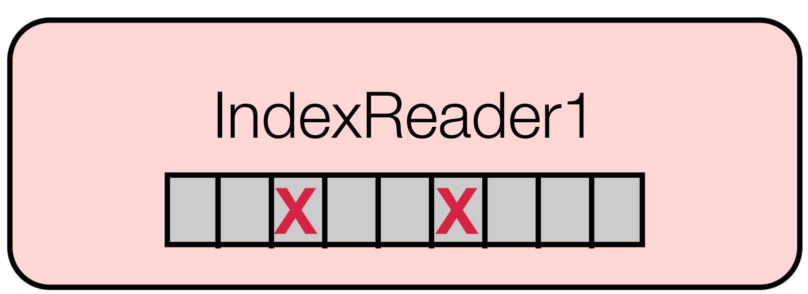

```
deleteDoc(2);
deleteDoc(5);
```

Deletes

- Today deletes are stored as BitSets

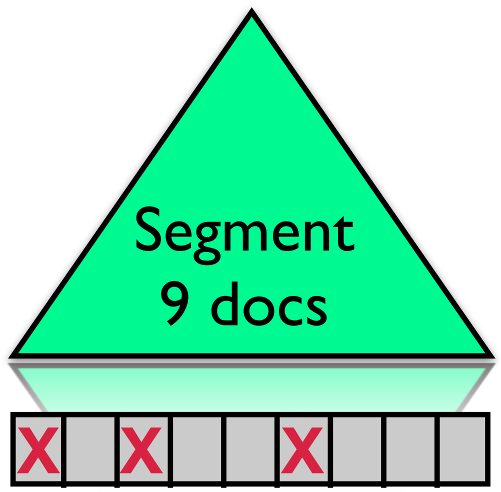



```
deleteDoc(2);
deleteDoc(5);
open IndexReader1
```

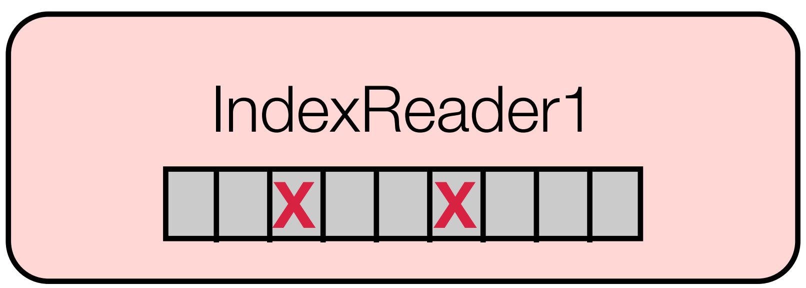


Deletes

- Today deletes are stored as BitSets

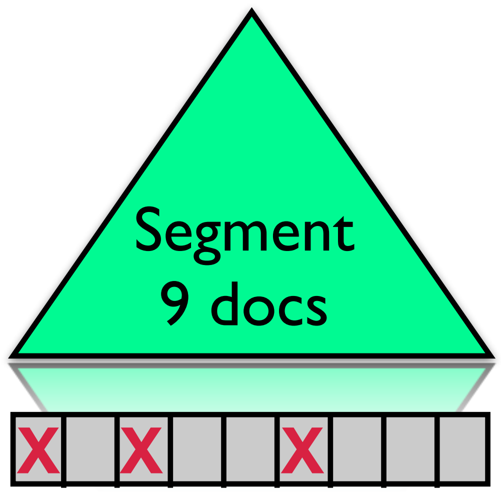



```
deleteDoc(2);
deleteDoc(5);
open IndexReader1
deleteDoc(0);
```

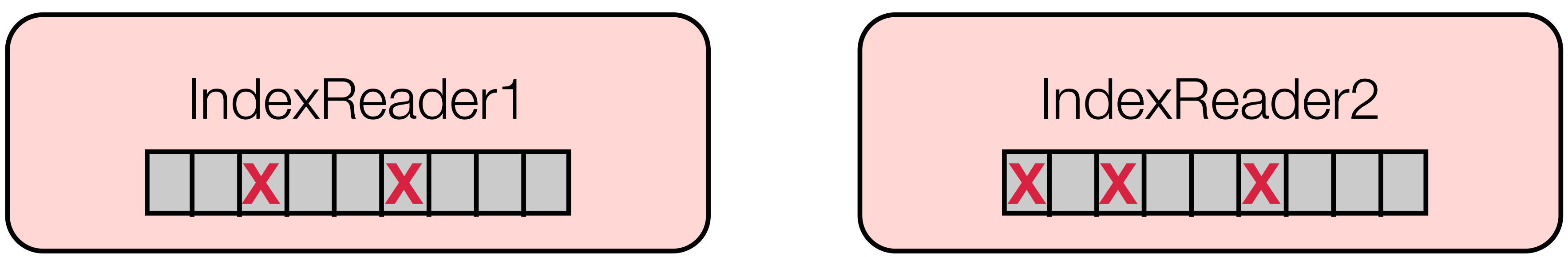


Deletes

- Today deletes are stored as BitSets

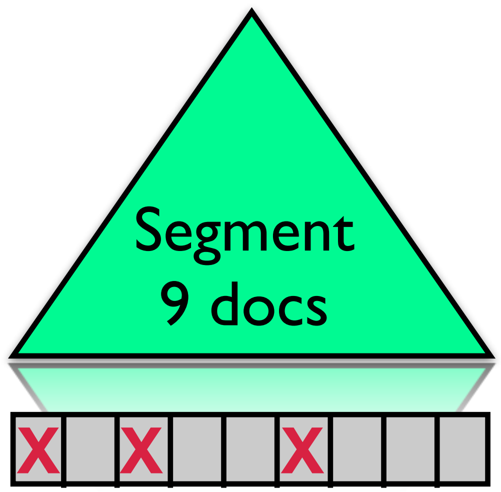



```
deleteDoc(2);
deleteDoc(5);
open IndexReader1
deleteDoc(0);
open IndexReader2
```

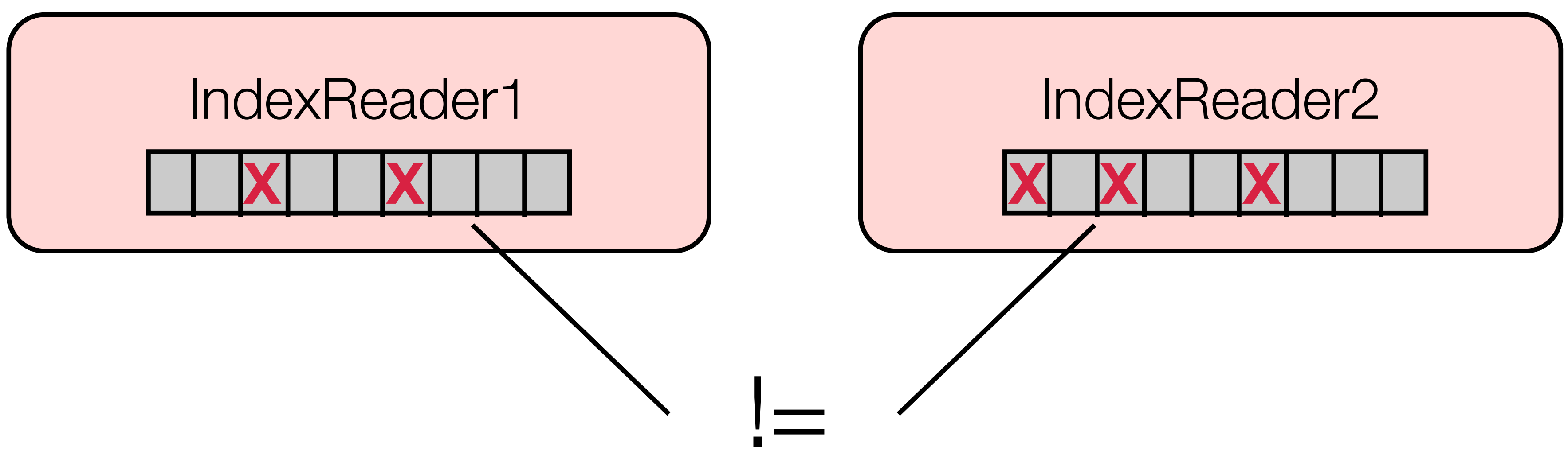


Deletes

- Today deletes are stored as BitSets

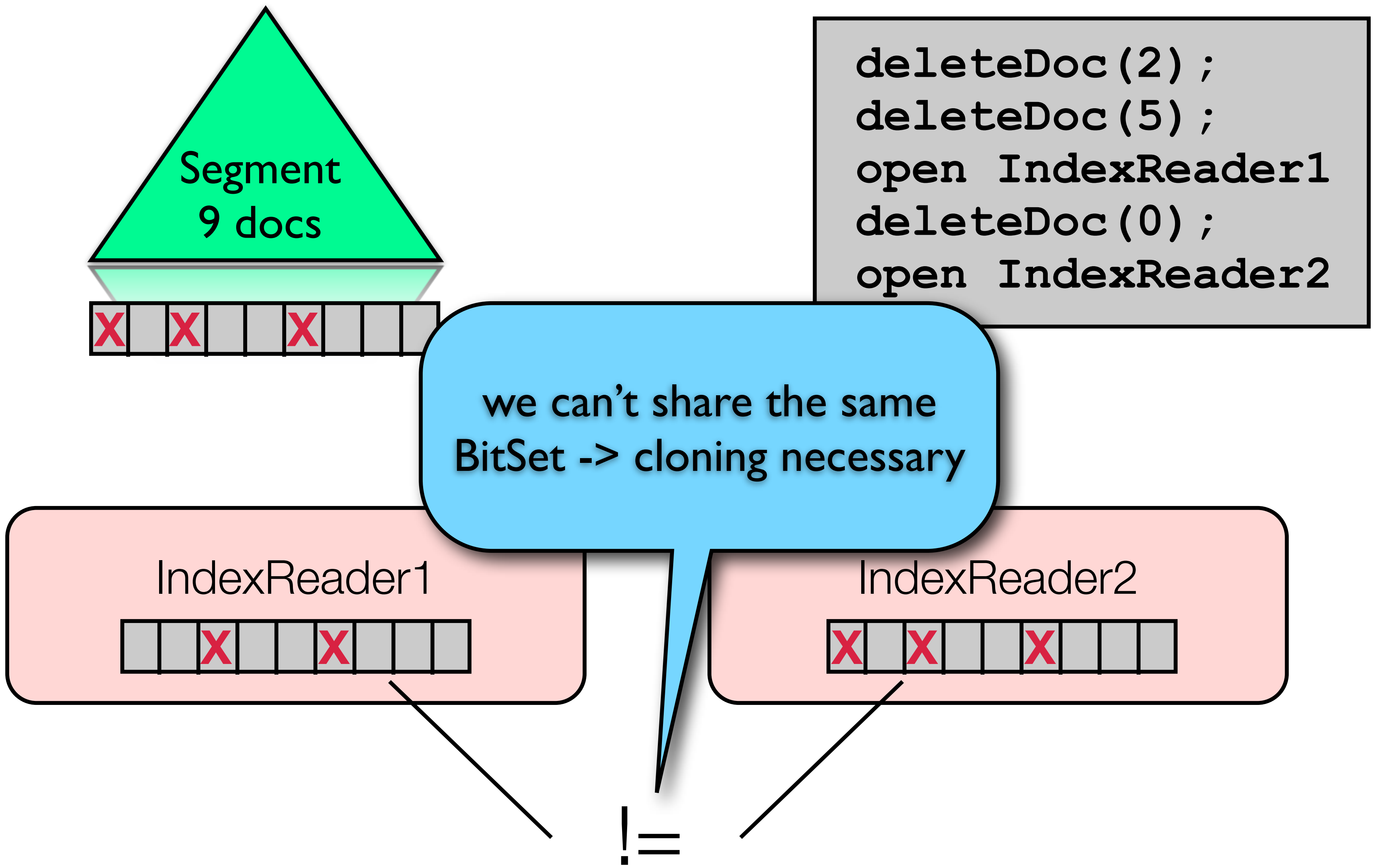



```
deleteDoc(2);
deleteDoc(5);
open IndexReader1
deleteDoc(0);
open IndexReader2
```



Deletes

- Today deletes are stored as BitSets

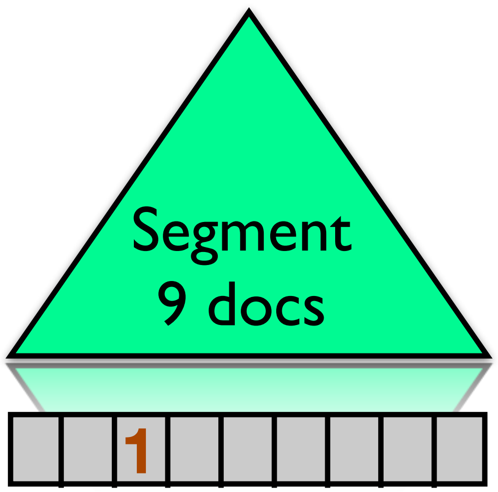


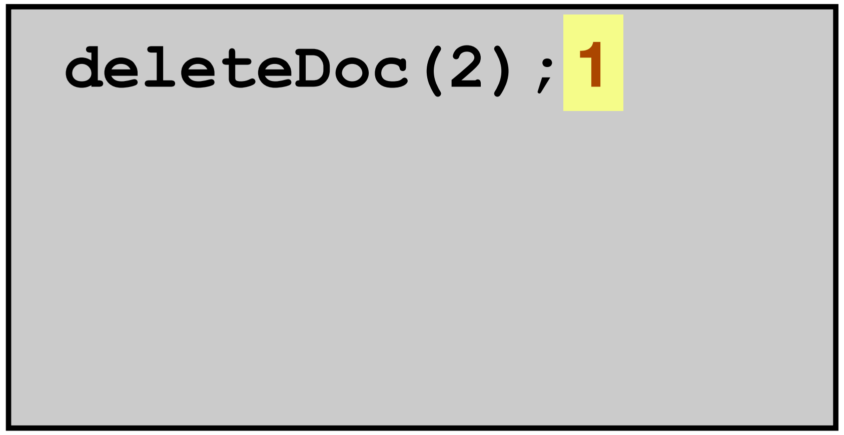
Deletes

- Each IndexReader may need its own copy of the BitSet
- Especially for large segments the cloning quickly becomes very inefficient, if deletes and IndexReader (re)opens are frequent
- Solution: Utilize sequence IDs instead of BitSets

Utilizing Sequence IDs for memory efficient deletes

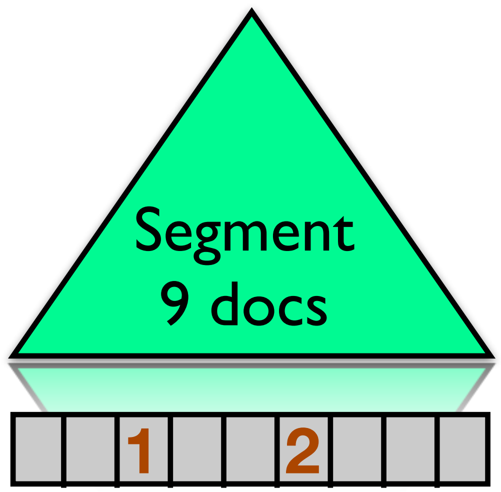
- LUCENE-2324: Use array of sequence IDs instead of BitSets





Utilizing Sequence IDs for memory efficient deletes

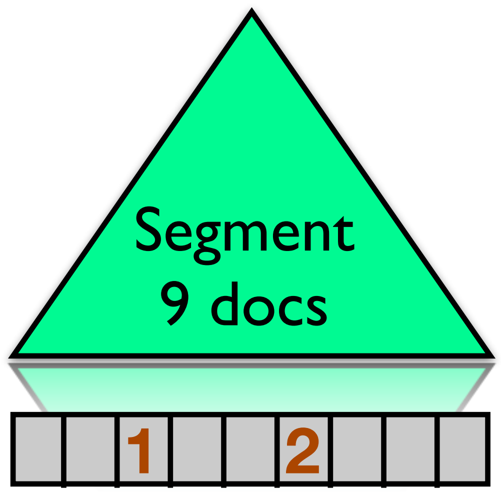
- LUCENE-2324: Use array of sequence IDs instead of BitSets



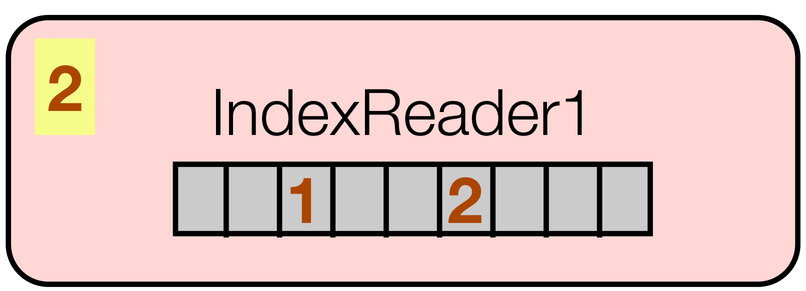

```
deleteDoc(2); 1
deleteDoc(5);   2
```

Utilizing Sequence IDs for memory efficient deletes

- LUCENE-2324: Use array of sequence IDs instead of BitSets

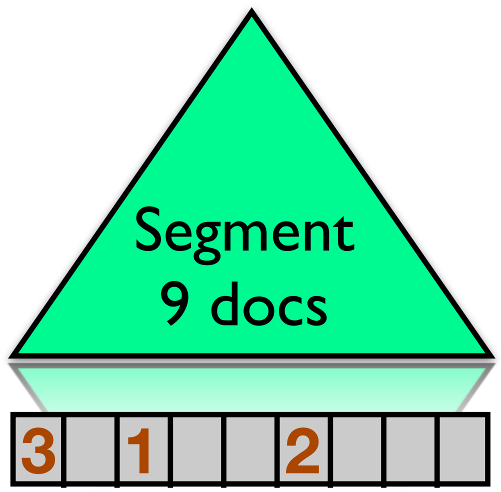



```
deleteDoc(2);  1
deleteDoc(5);    2
open IndexReader1
```

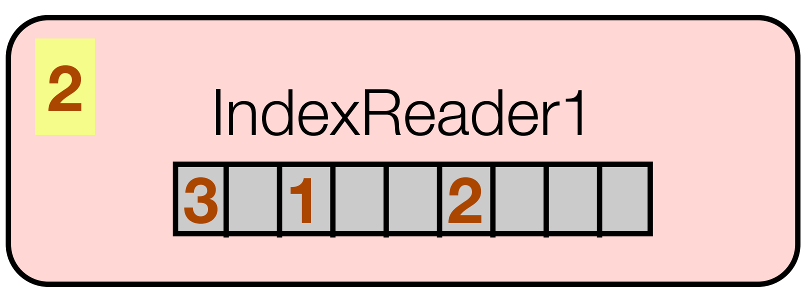


Utilizing Sequence IDs for memory efficient deletes

- LUCENE-2324: Use array of sequence IDs instead of BitSets

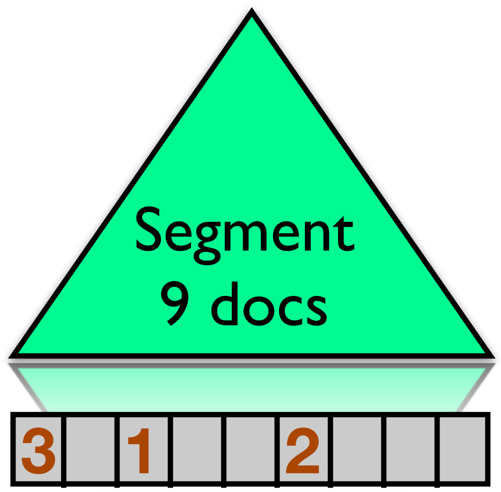



```
deleteDoc(2); 1
deleteDoc(5);   2
open IndexReader1
deleteDoc(0); 3
```

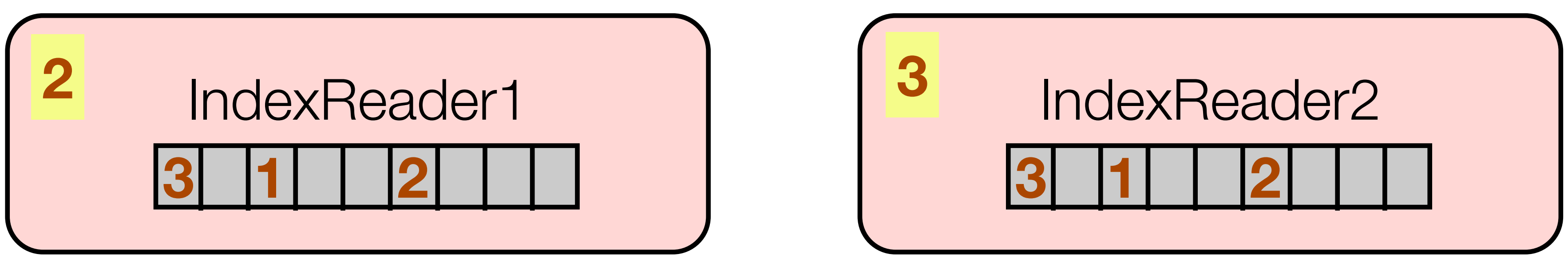


Utilizing Sequence IDs for memory efficient deletes

- LUCENE-2324: Use array of sequence IDs instead of BitSets

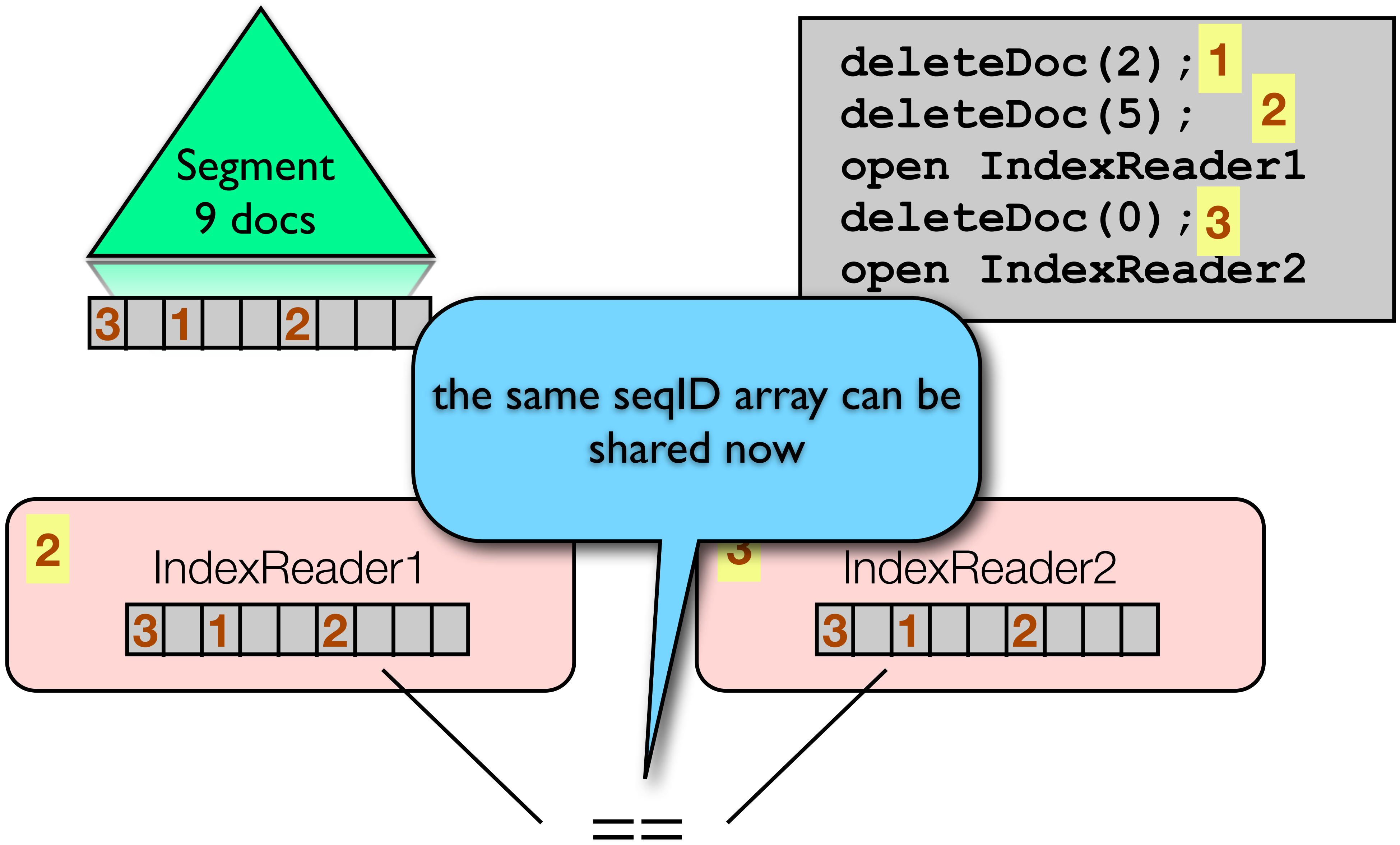



```
deleteDoc(2); 1
deleteDoc(5);   2
open IndexReader1
deleteDoc(0); 3
open IndexReader2
```



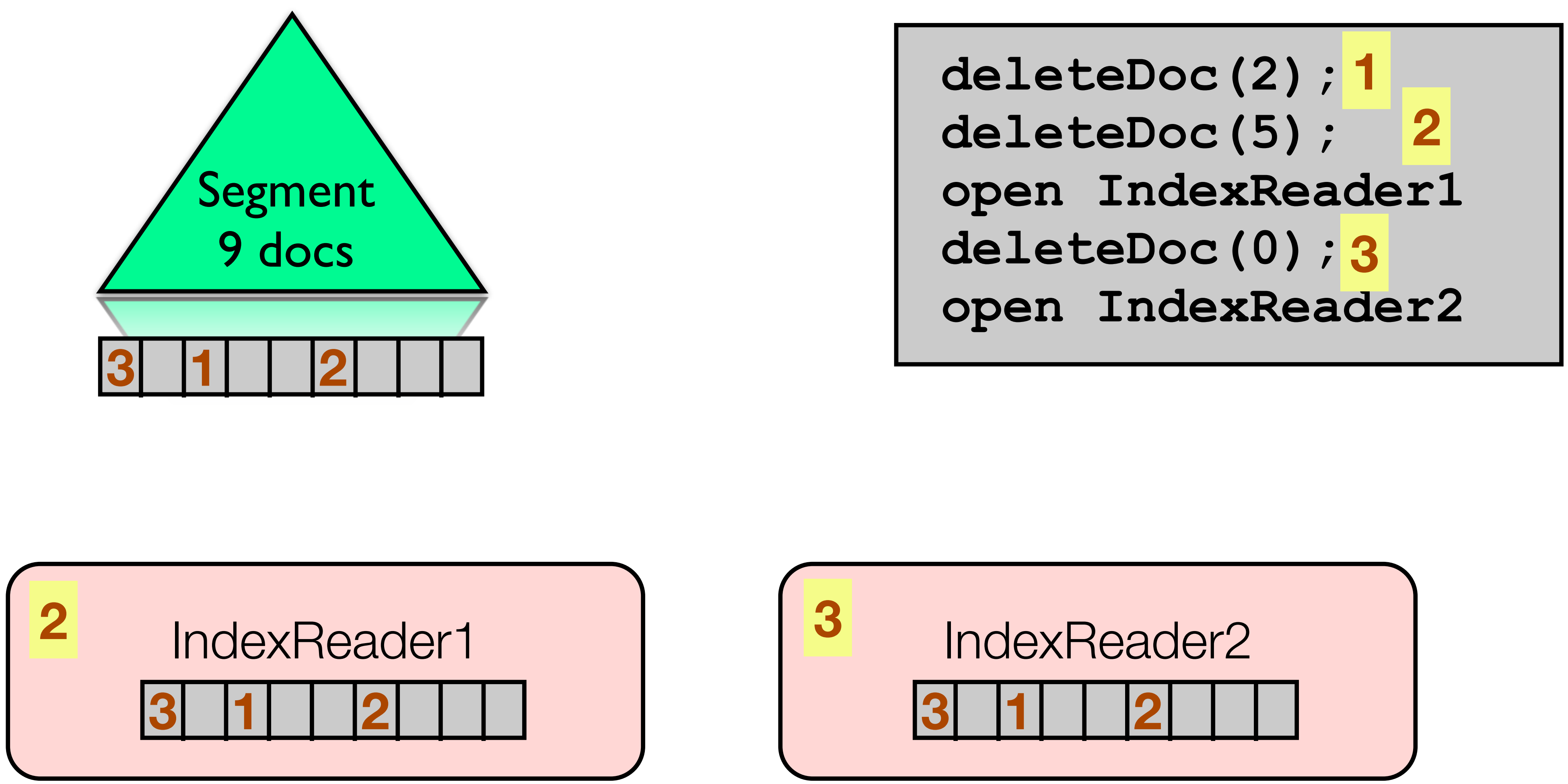
Utilizing Sequence IDs for memory efficient deletes

- LUCENE-2324: Use array of sequence IDs instead of BitSets



Utilizing Sequence IDs for memory efficient deletes

- LUCENE-2324: Use array of sequence IDs instead of BitSets




```
boolean isDeleted = (seqId[doc] <= readerSeqID);

Reader1: seqId[0] = 3, readerSeqID = 2 -> isDeleted = false
Reader2: seqId[0] = 3, readerSeqID = 3 -> isDeleted = true
```

Utilizing Sequence IDs for memory efficient deletes

- No cloning necessary anymore
- Memory consumption for deletes does not increase when many IndexReaders are opened

Goal 3:
Opening a RAM IndexReader should be so cheap, so that a new reader can be opened for every query (drops latency close to zero)

Using sequence IDs for document tracking

- Lucene's IndexWriter handles two kinds of exceptions: Aborting exceptions (e.g. OutOfMemoryError) and non-aborting exceptions (e.g. document encoding problem)

- When an aborting exception occurs, then the IndexWriter tries to commit all docs to the index that were successfully flushed before the error occurred

- Problem: Today it's not possible to know which documents made it into the index and which ones were dropped due to the error. **Which docs do I have to reindex?**

- Solution: `IndexWriter.commit()` will also return the sequence ID of the last write operation (add, delete, update) that was committed

Using sequence IDs for document tracking

- An external log can be used to replay all operations that were lost due to the aborting exception
- It's easy to find out which write operations need to be replayed by checking the sequence ID that `commit()` returns

Realtime Search with Lucene

Agenda

- Introduction
- Near-realtime Search (NRT)
- Searching DocumentsWriter's RAM buffer
- Sequence IDs
- **Twitter prototype**
- Roadmap

Twitter prototype

Postinglist format

- Tweets are only 140 chars long

- Use 32-bit integers for postings: 24 bits for the docID (max segment size is 16.7M docs), 8 bits for the position (position can only have values 0-255; enough for tweets)

- Decoding speed significantly improved compared to delta and VInt decoding (early experiments suggest 5x improvement compared to vanilla Lucene with FSDirectory)

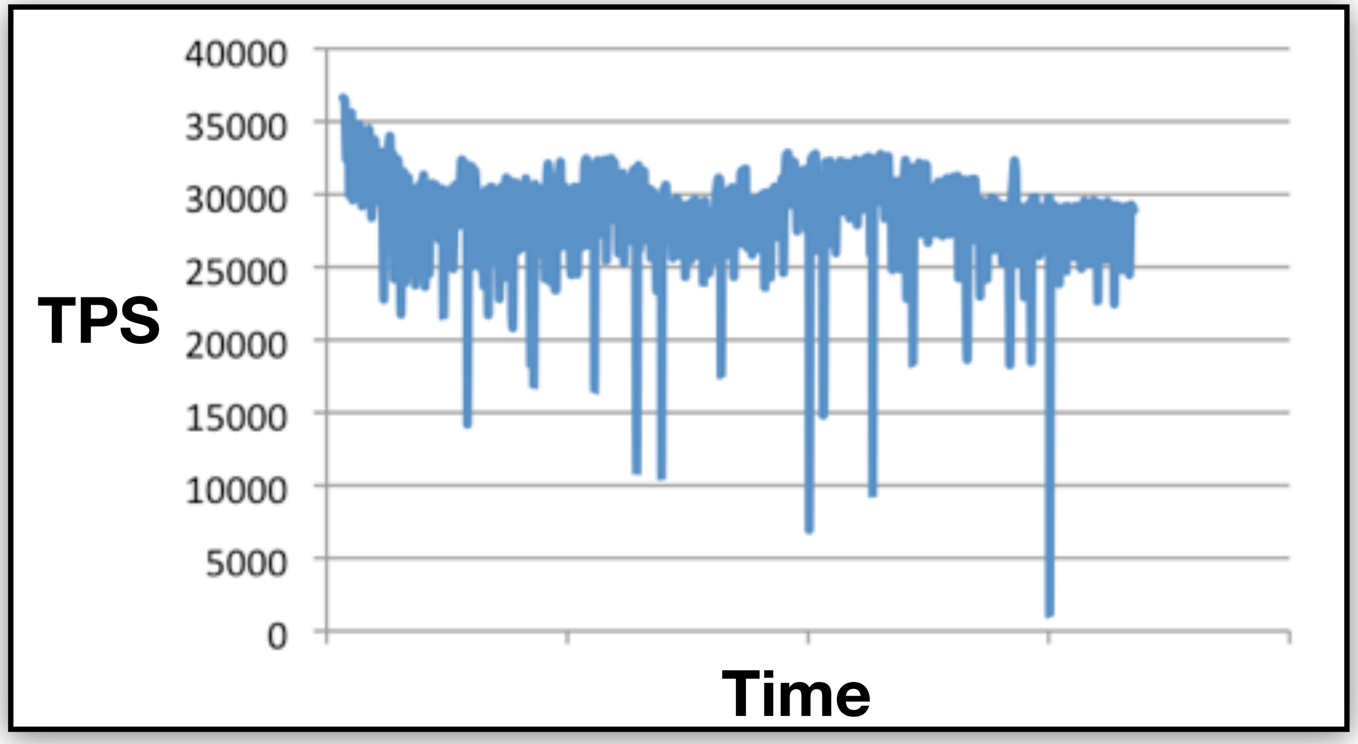
- In-memory postinglists can be traversed in reverse order -> early termination if time is a dominant factor of ranking score (as it usually is in realtime search)

Early performance experiments

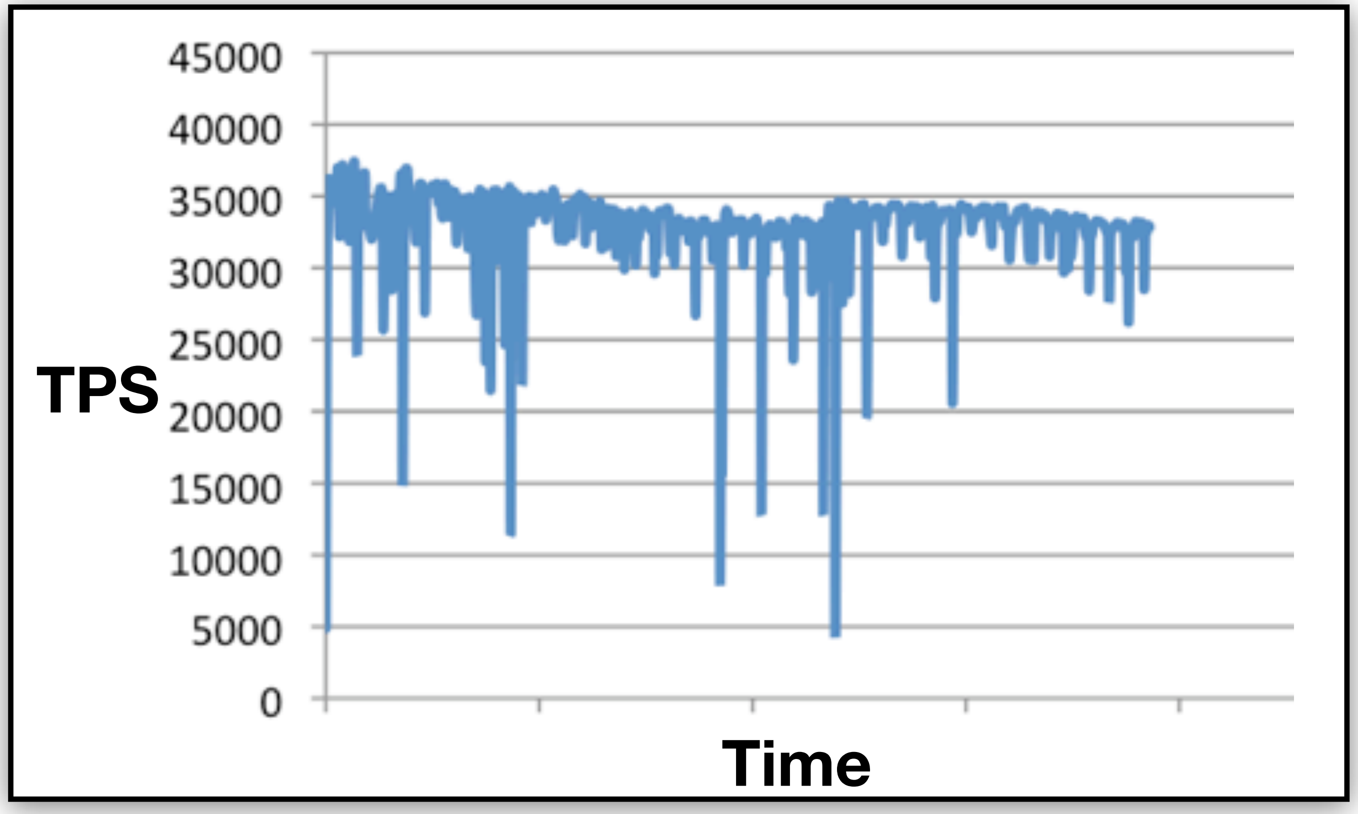
- On a single machine we can (without much tuning yet):
    - Index ~60,000 tweets/sec (very simple text analysis in the prototype)
    - Search with ~15,000-20,000 queries/sec
    - Lock-free algorithm: Results show, that indeed indexing and search performance are independent

Early performance experiments

Indexing with one thread **while querying with multiple threads**

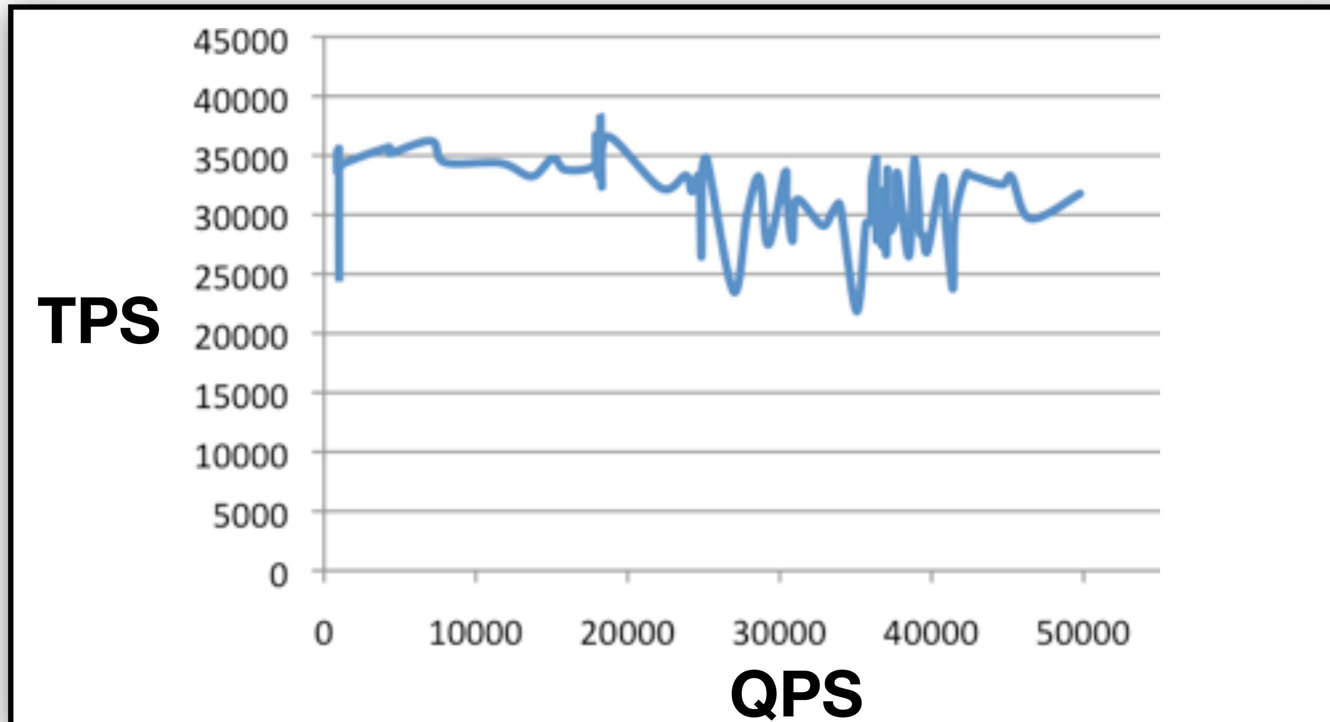


Only indexing with one thread



Early performance experiments

Indexing performance over varying query load



- No "trend" here: indexing performance pretty much independent of query load

- TPS goes down only if more threads are used than CPU cores are present, because thread scheduling becomes expensive

Goal 2:
Maintain high indexing performance with large RAM buffer, and independent of the query load

Realtime Search with Lucene

Agenda

- Introduction
- Near-realtime Search (NRT)
- Searching DocumentsWriter's RAM buffer
- Sequence IDs
- Twitter prototype
- Roadmap

Roadmap

Roadmap

- ~~LUCENE-2329: Parallel posting arrays~~
- LUCENE-2324: Per-thread DocumentsWriter and sequence IDs
- LUCENE-2346: Change in-memory postinglist format
- LUCENE-2312: Search on DocumentsWriters RAM buffer
- IndexReader, that can switch from RAM buffer to flushed segment on-the-fly
- Sorted term dictionary (wildcards, numeric queries)
- Stored fields, TermVectors, Payloads (Attributes)

Questions?

Realtime Search with Lucene

Michael Busch
@michibusch
michael@twitter.com
buschmi@apache.org