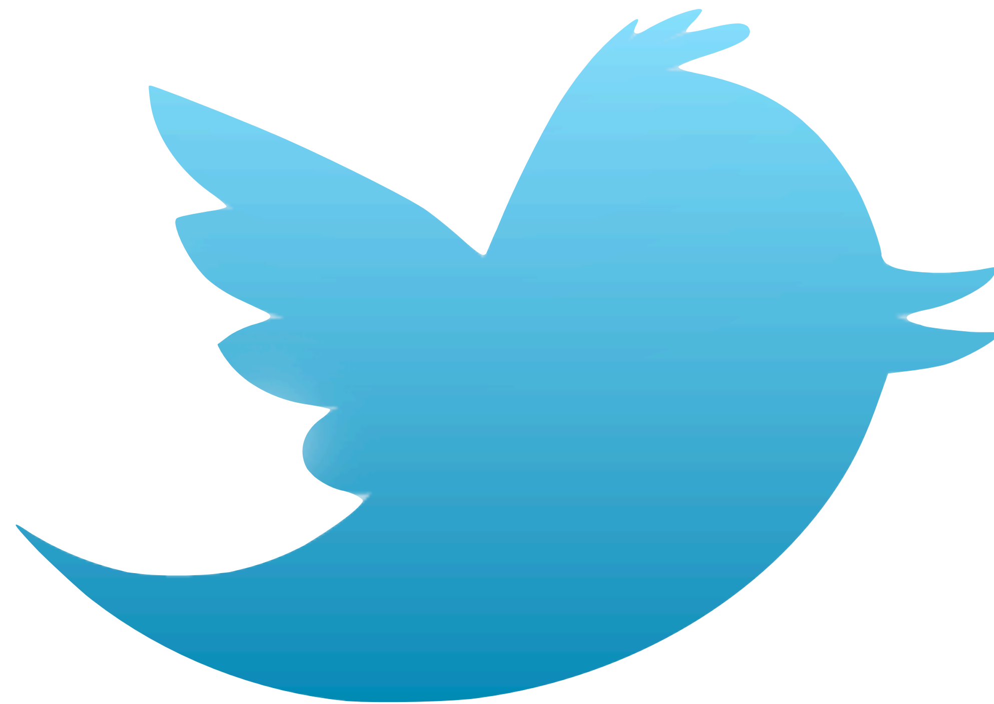
A Billion Queries Per Day

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



A Billion Queries Per Day

Agenda

- Introduction
- Posting list format and early query termination
- Index memory model
- Lock-free algorithms and data structures
- Roadmap

Introduction

Introduction

- Search @ Twitter today:
    - >1,000 TPS (tweets/sec) = >80,000,000 tweets/day
    - >12,000 QPS (queries/sec) = >1,000,000,000 queries/day
    - <10s latency (entire pipeline)
    - <1s latency (indexer)
- Performance goal: Ability to scale to support Twitter’s continued growth

Introduction

- 1st gen search engine based on MySQL
- Next-gen engine based on (partially rewritten) Lucene Java
- Significantly more performant (orders of magnitude)
- Much easier to develop new features on the new platform - stay tuned!

A Billion Queries Per Day

Agenda

- Introduction
- Posting list format and early query termination
- Index memory model
- Lock-free algorithms and data structures
- Roadmap

Posting list format and early query termination

Objectives

- Only evaluate as few documents as possible before terminating the query
- Rank documents in reverse time order (newest documents first)

Inverted Index 101

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 101

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 101

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 101

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

Early query termination

Query: old

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

Early query termination

Query: old

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
in	5	<
keep	3	<
keeper	3	<
keeps	3	<1
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>

Walk posting list in reverse order

Dictionary and posting lists

Early query termination

Query: old

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
in	5	<
keep	3	<
keeper	3	<
keeps	3	<1
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>

E.g. 2 results are requested: Optimal termination after exactly two postings were read

Dictionary and posting lists

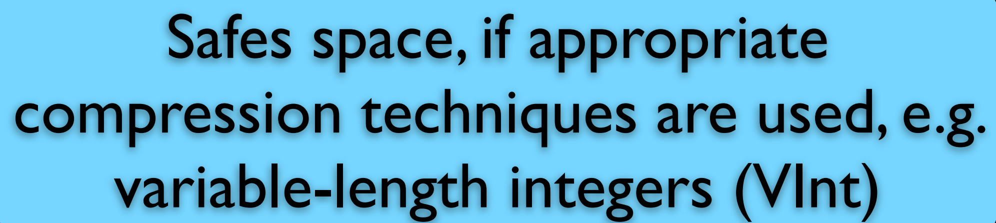
Early query termination

- Possible if time-ordered results are desired (which is often the case in realtime search applications)
- Posting list requirements:
    - Pointer from dictionary to **last** posting in a list (must always be up to date)
    - Possibility to iterate lists in reverse order
    - Allow efficient skipping

Posting list encodings

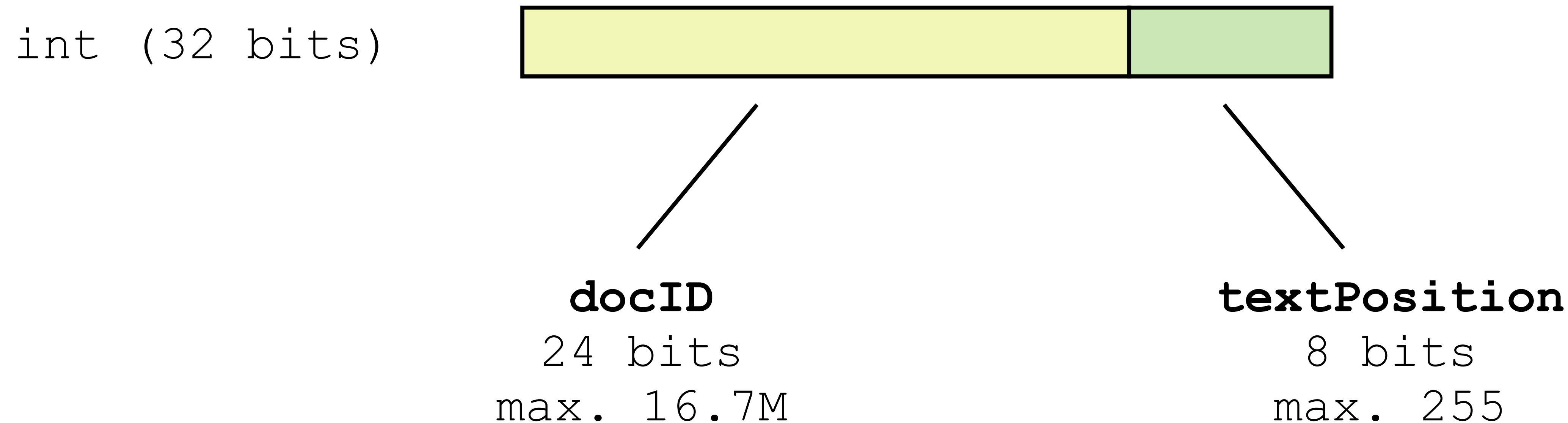
Doc IDs to encode: 5, 15, 9000, 9002

Delta encoding: 5, 10, 8985, ***2***



- Lucene uses a combination of delta encoding and VInt compression
- VInts are expensive to decode
- Problem 1: How to traverse posting lists backwards?
- Problem 2: How to write a posting atomically, if not a primitive java type

Postinglist format



- Tweet text can only have 140 chars
- Decoding speed significantly improved compared to delta and VInt decoding (early experiments suggest 5x improvement compared to vanilla Lucene with FSDirectory)

Postinglist format

- ints can be written atomically in Java
- Backwards traversal easy on absolute docIDs (not deltas)
- Every posting is a possible entry point for a searcher
- Skipping can be done without additional data structures as binary search, even though there are better approaches which should be explored
- On tweet indexes we need about 30% more storage for docIDs compared to delta+Vints, but that’s compensated by savings on position encoding!
- Max. segment size: 2^24 = 16.7M tweets

Objectives

- Only evaluate as few documents as possible before terminating the query ✔
- Rank documents in reverse time order (newest documents first) ✔

A Billion Queries Per Day

Agenda

- Introduction
- Posting list format and early query termination
- ‣ Index memory model
- Lock-free algorithms and data structures
- Roadmap

Index memory model

Objectives

- Store large amount of linked lists of a priori unknown lengths efficiently in memory
- Allow traversal of lists in reverse order (backwards linked)
- Allow efficient garbage collection

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

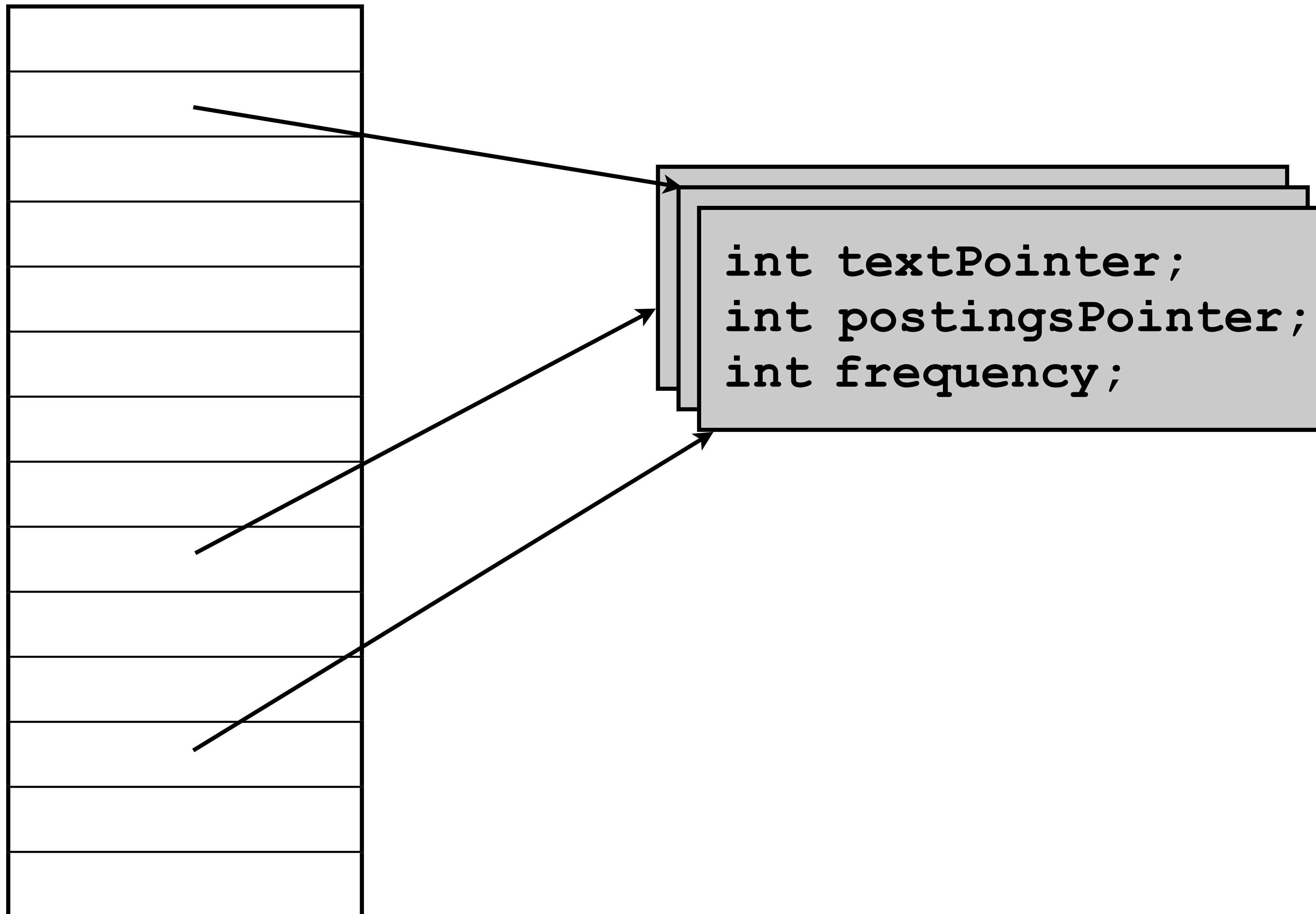
LUCENE-2329: Parallel posting arrays

```
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

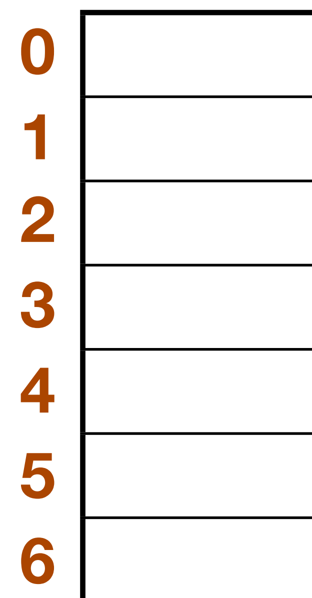
LUCENE-2329: Parallel posting arrays



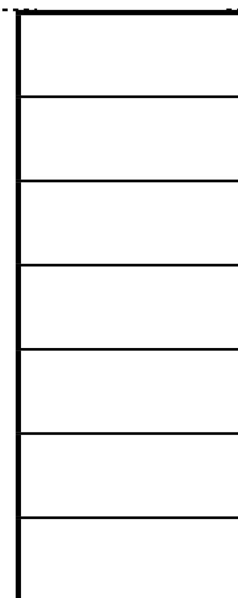
LUCENE-2329: Parallel posting arrays

termID
int[]

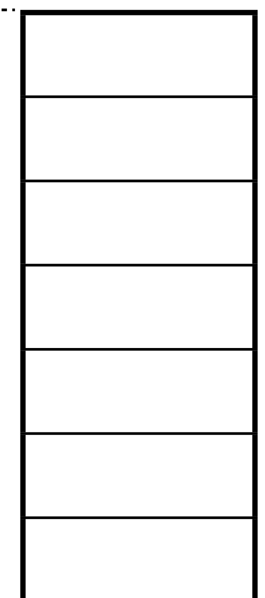
textPointer;
int[]



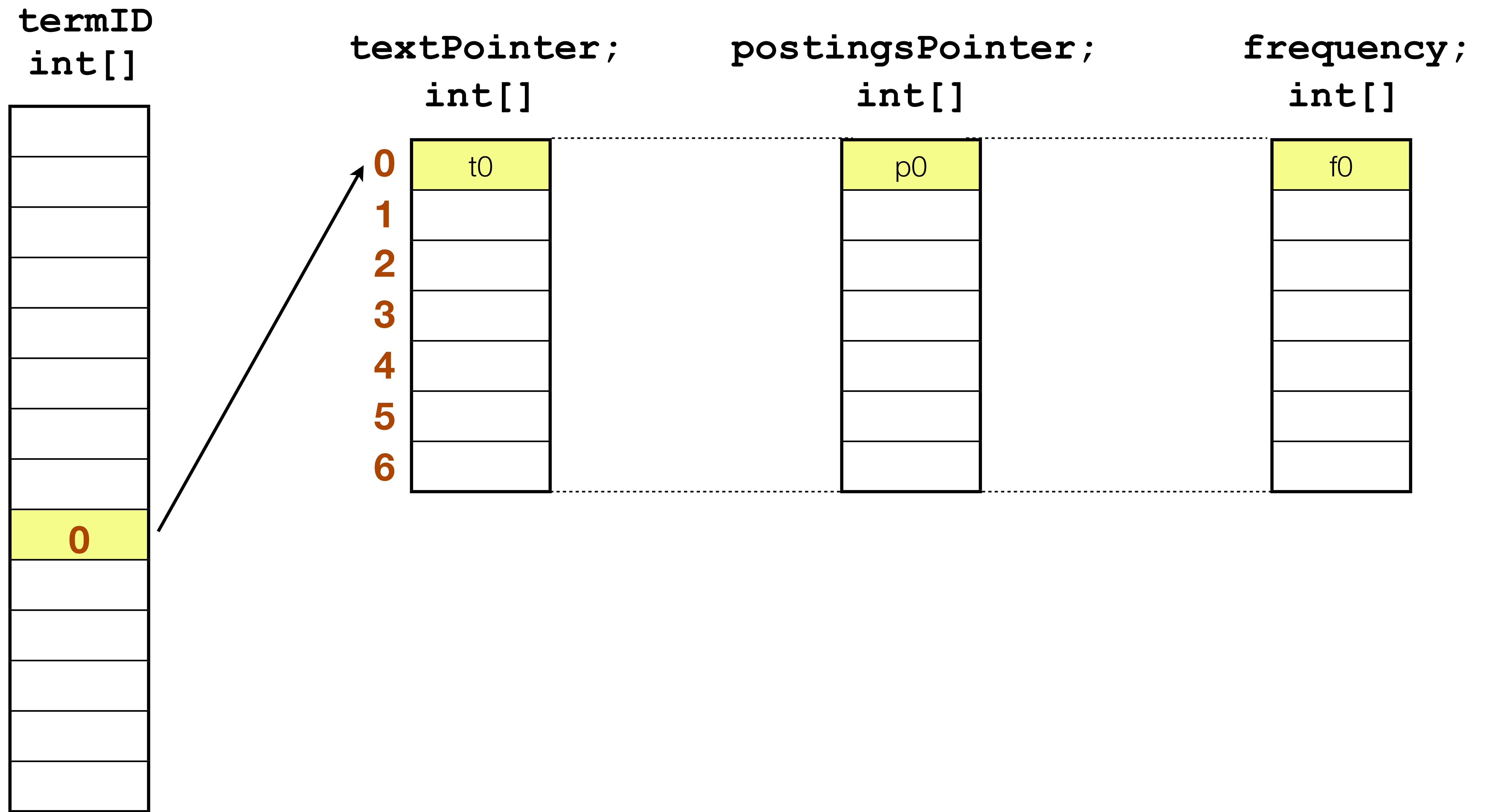
postingsPointer;
int[]



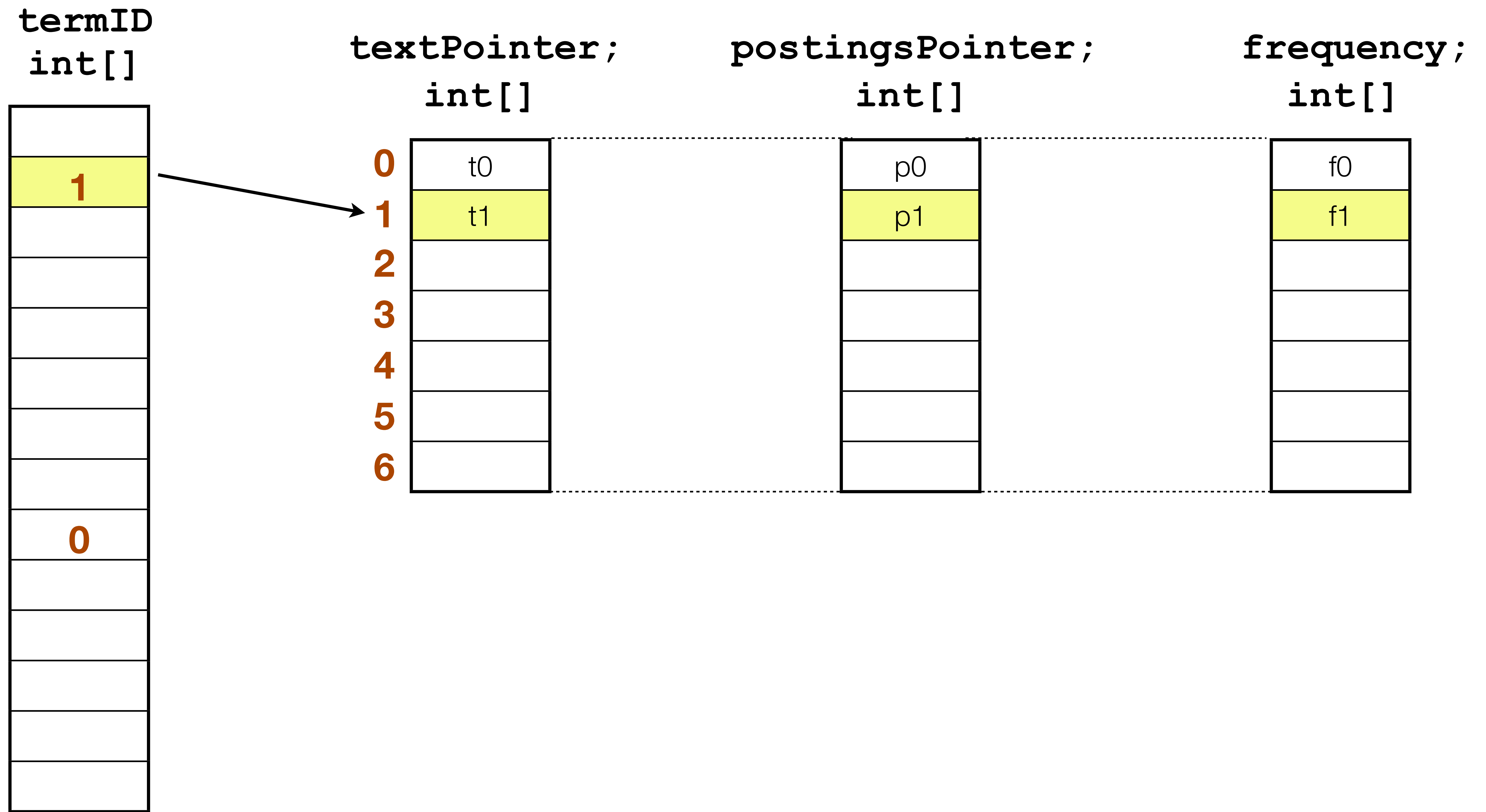
frequency;
int[]



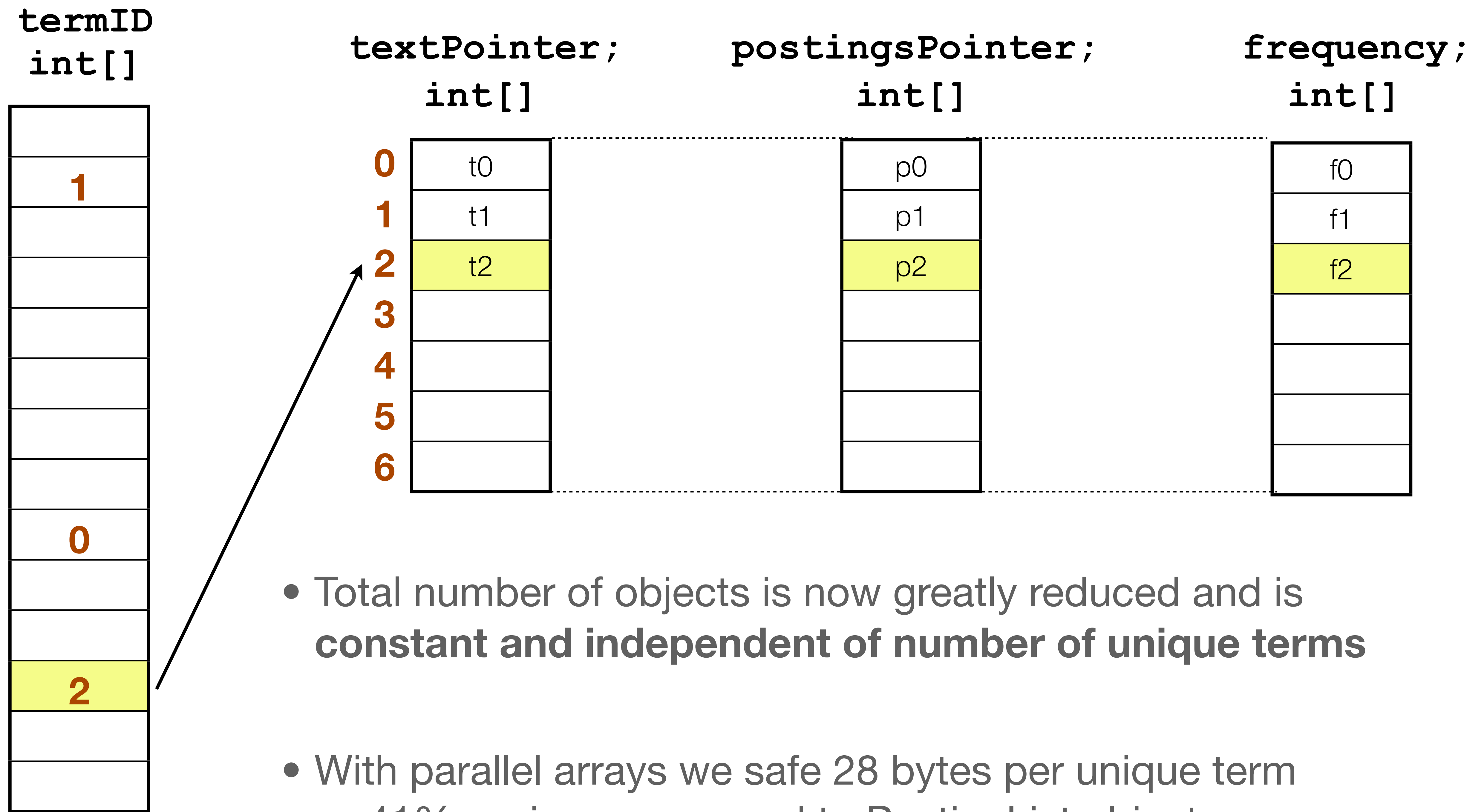
LUCENE-2329: Parallel posting arrays



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

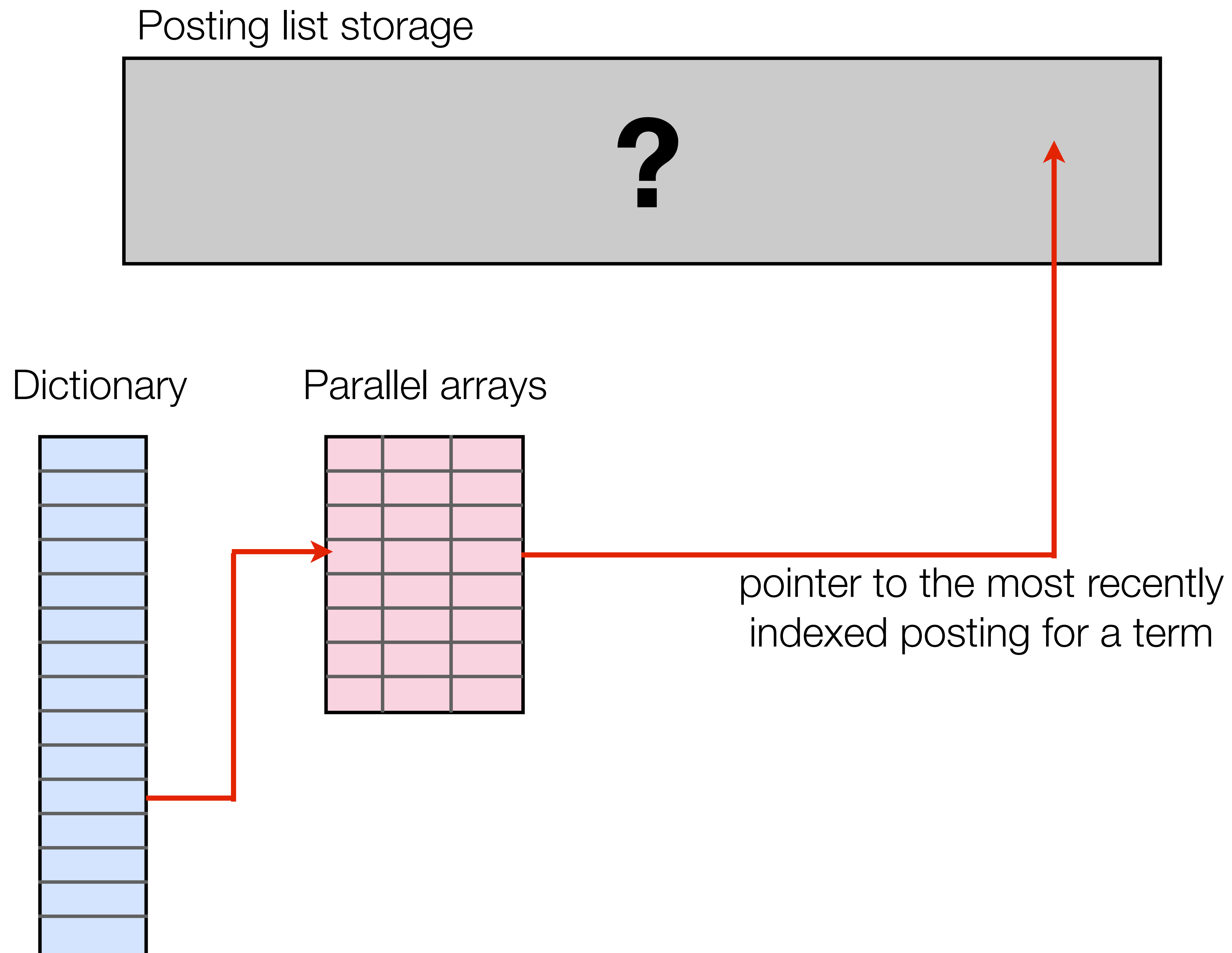
LUCENE-2329: Parallel posting arrays - Performance

- 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

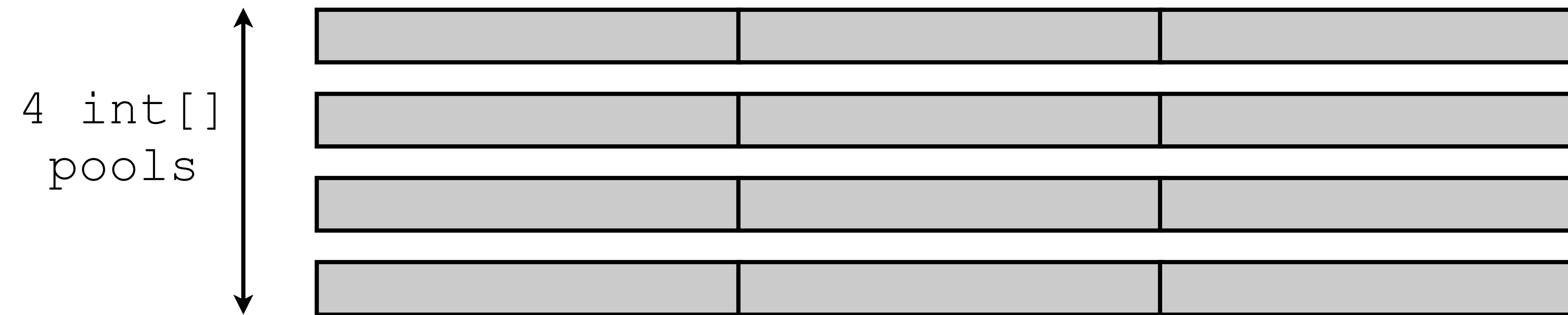
Inverted index components



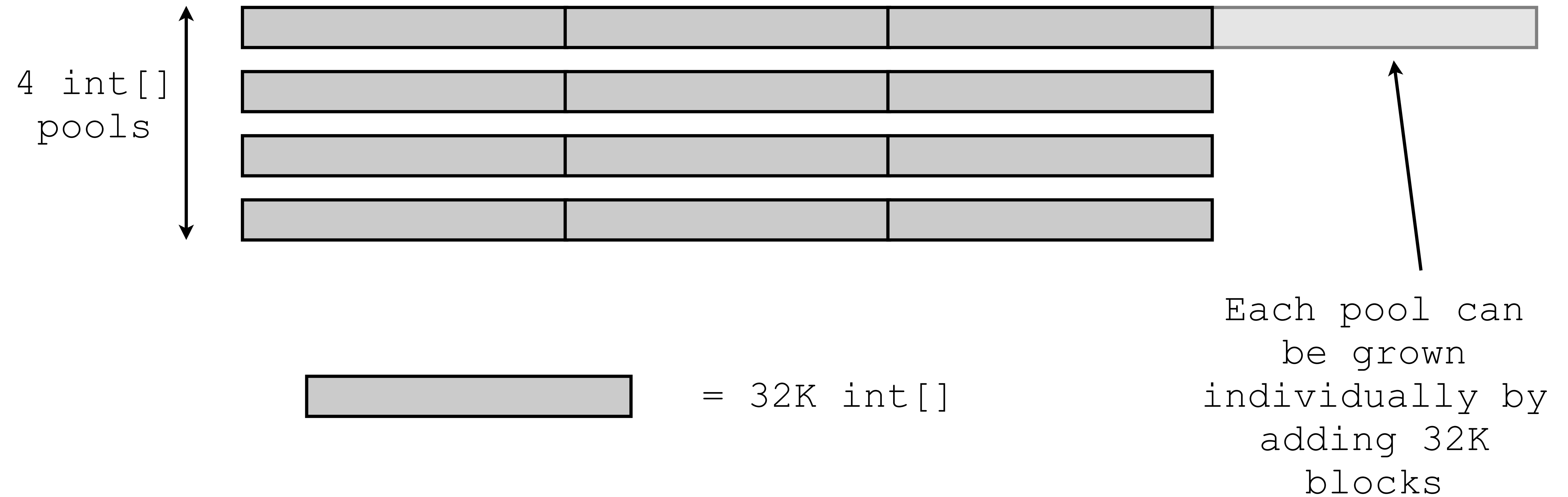
Posting lists storage - Objectives

- Store many single-linked lists of different lengths space-efficiently
- The number of java objects should be independent of the number of lists or number of items in the lists
- Every item should be a possible entry point into the lists for iterators, i.e. items should not be dependent on other items (e.g. no delta encoding)
- Append and read possible by multiple threads in a lock-free fashion (single append thread, multiple reader threads)
- Traversal in backwards order

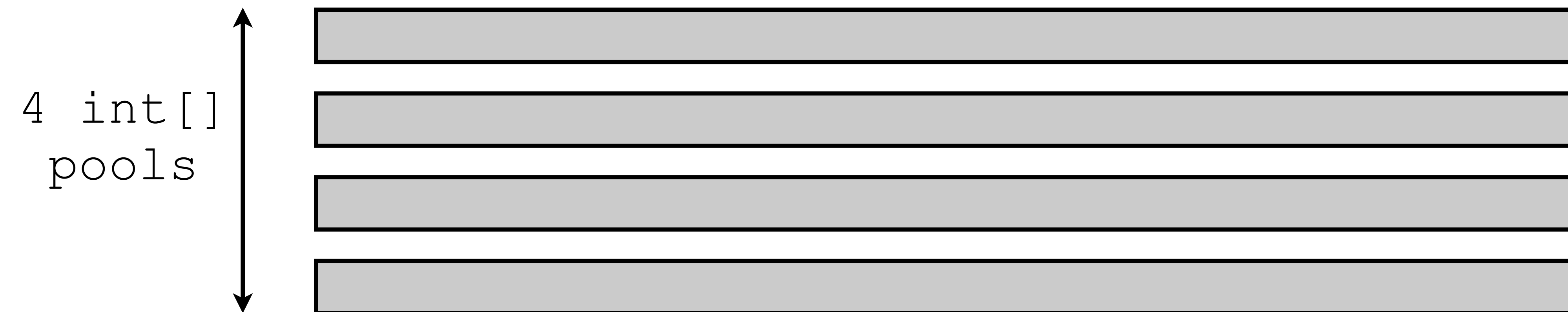
Memory management



Memory management

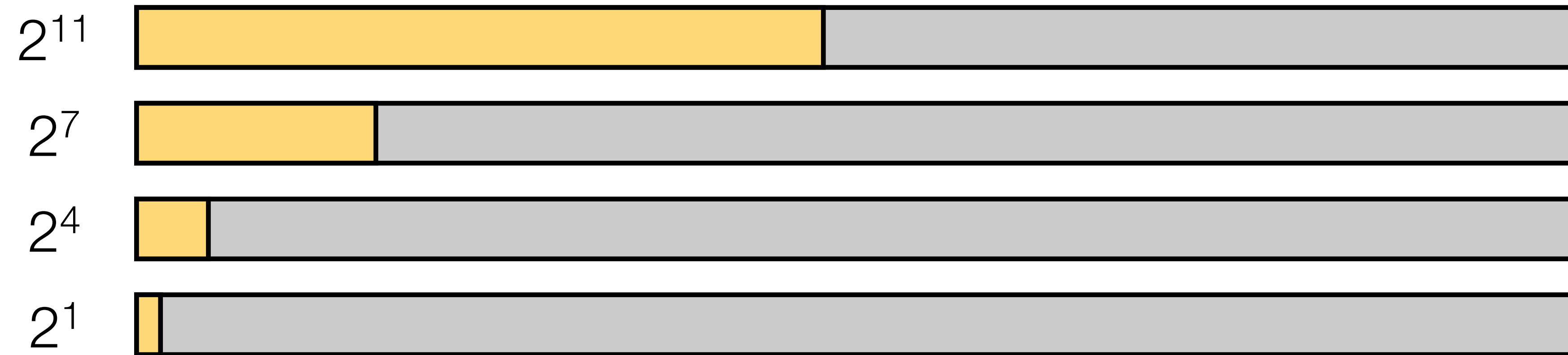


Memory management



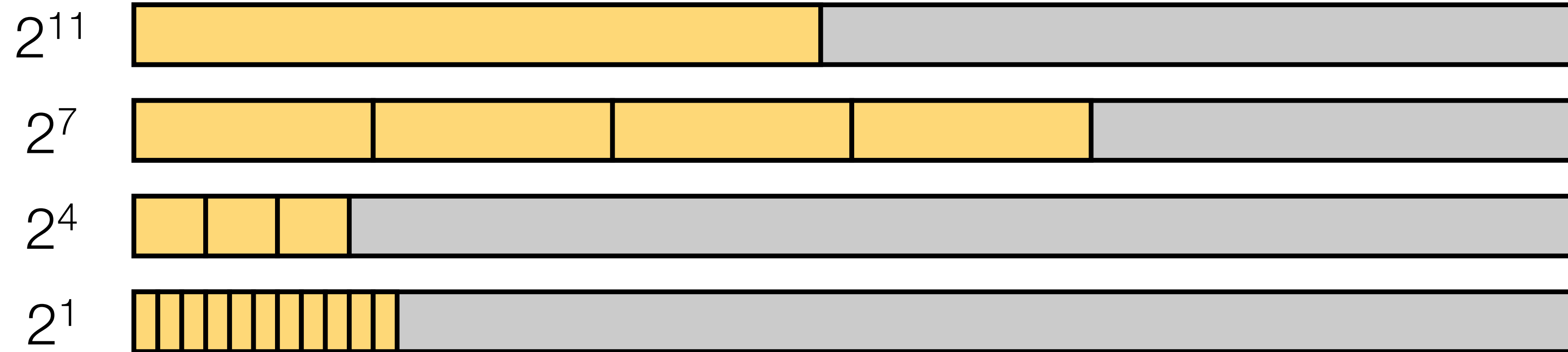
- For simplicity we can forget about the blocks for now and think of the pools as continuous, unbounded int[] arrays
- Small total number of Java objects (each 32K block is one object)

Memory management

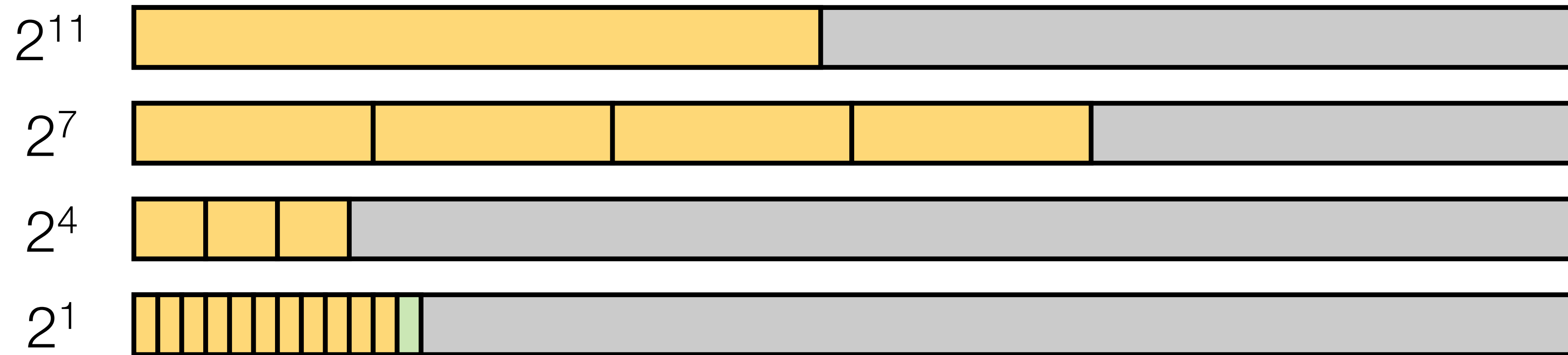


- Slices can be allocated in each pool
- Each pool has a different, but fixed slice size

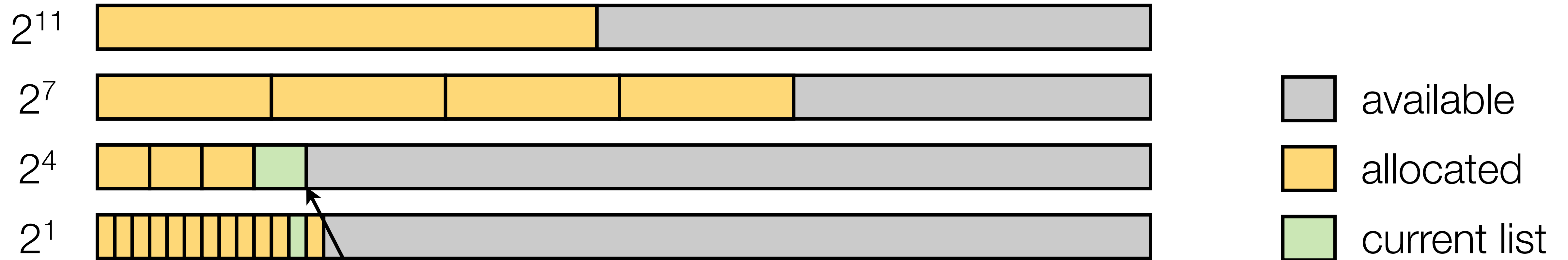
Adding and appending to a list



Adding and appending to a list

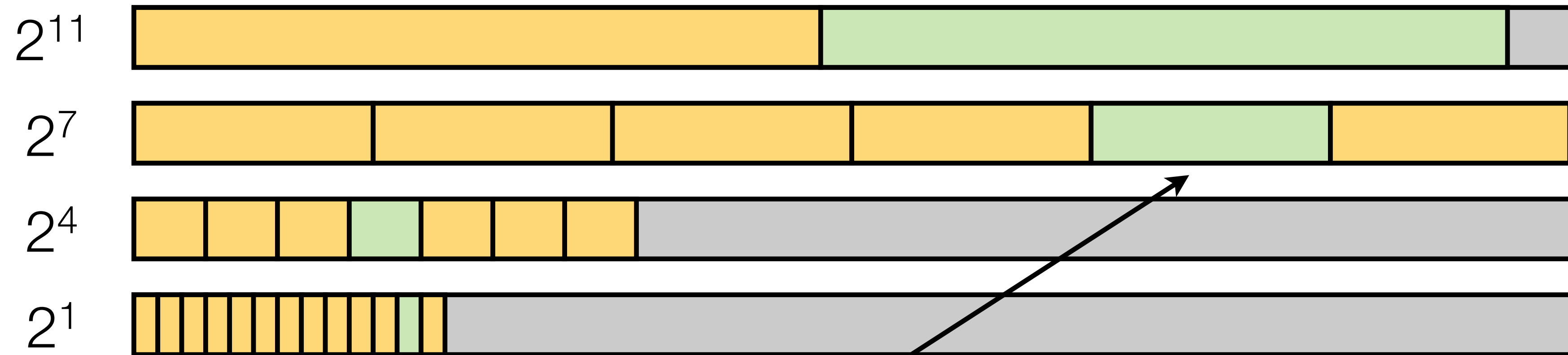


Adding and appending to a list



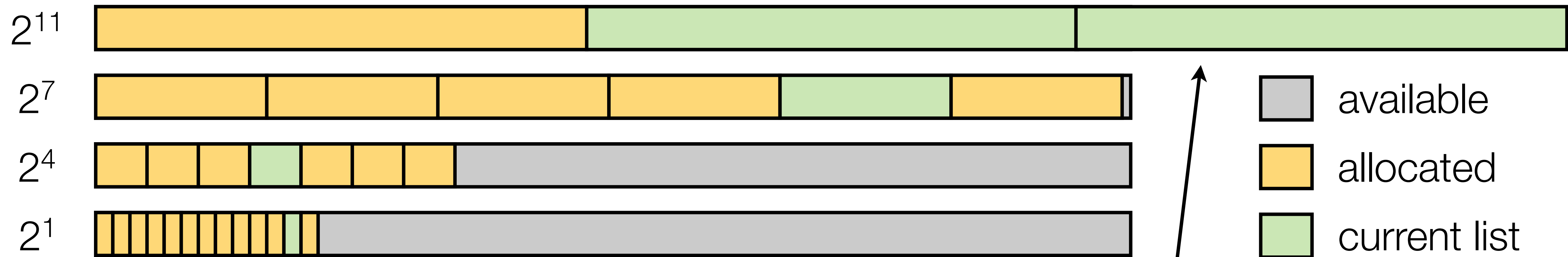
When first slice is full, allocate another one in second pool

Adding and appending to a list



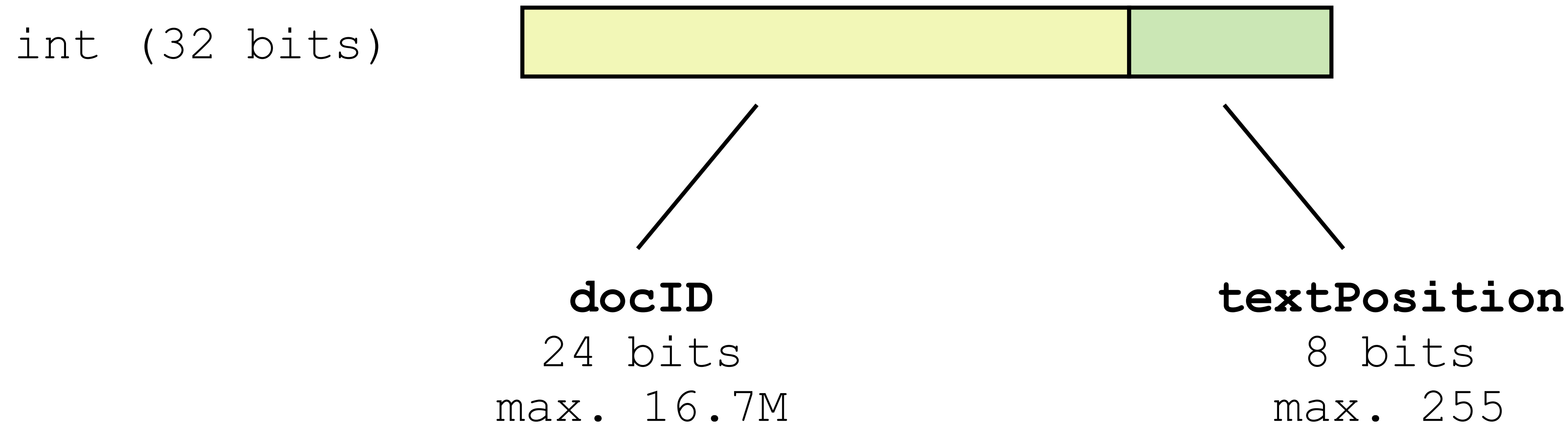
Allocate a slice on each level as list grows

Adding and appending to a list



On upper most level one list can own multiple slices

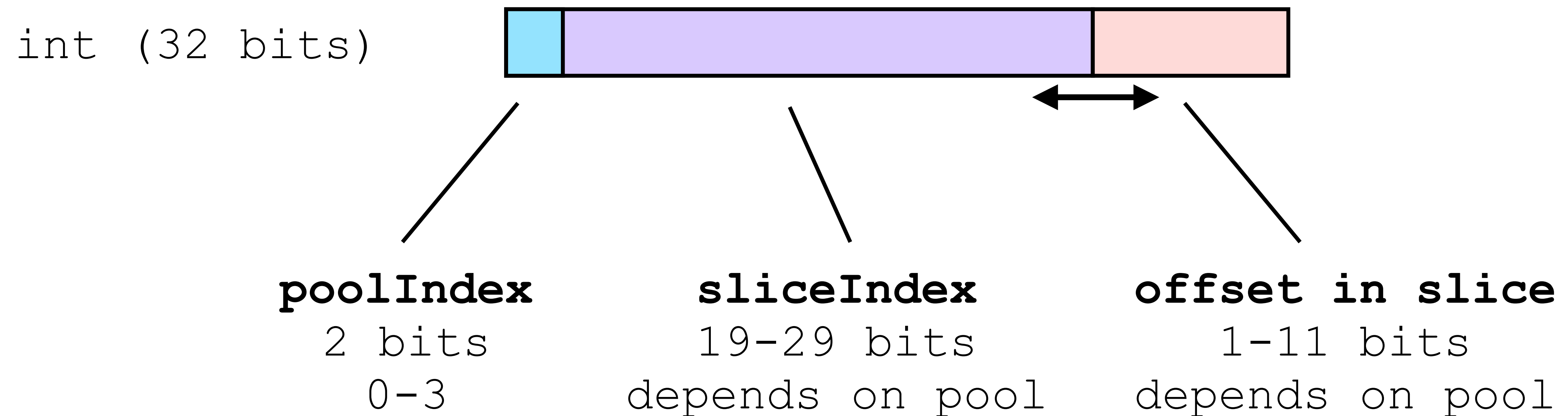
Posting list format



- Tweet text can only have 140 chars
- Decoding speed significantly improved compared to delta and VInt decoding (early experiments suggest 5x improvement compared to vanilla Lucene with FSDirectory)

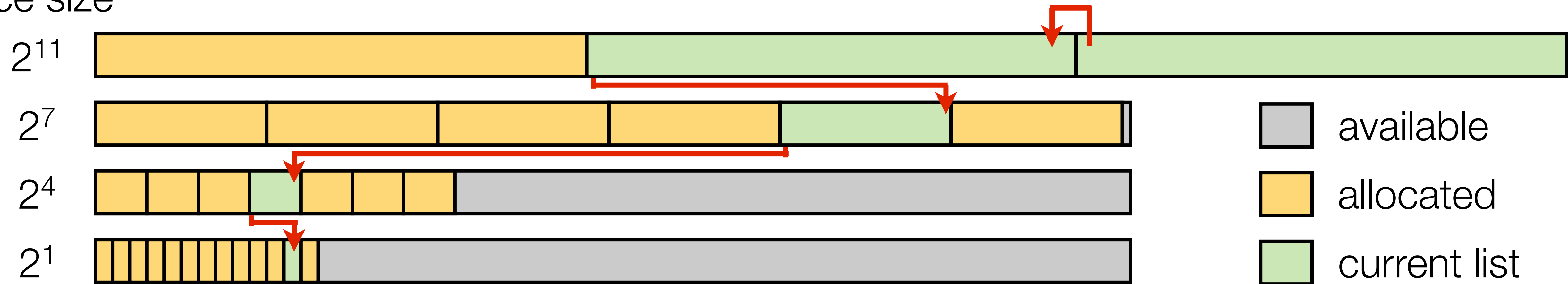
Addressing items

- Use 32 bit (int) pointers to address any item in any list unambiguously:



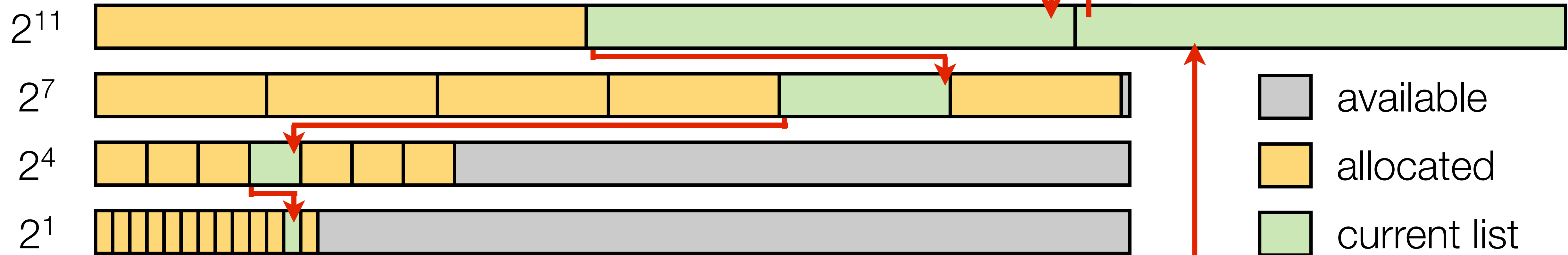
- Nice symmetry: Postings and address pointers both fit into a 32 bit int

Linking the slices

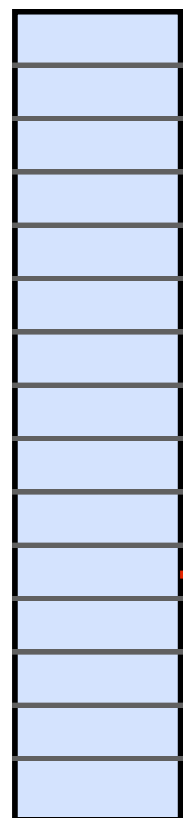


Linking the slices

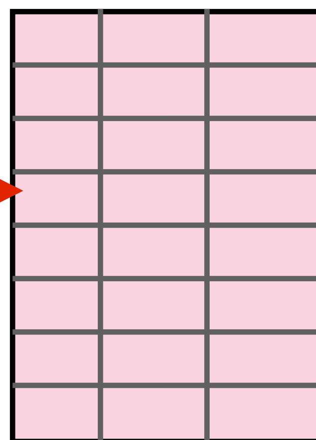
slice size



Dictionary



Parallel arrays



pointer to the last posting indexed for a term

Objectives

- Store large amount of linked lists of a priori unknown lengths efficiently in memory ✓
- Allow traversal of lists in reverse order (backwards linked) ✓
- Allow efficient garbage collection ✓

A Billion Queries Per Day

Agenda

- Introduction
- Posting list format and early query termination
- Index memory model
- ‣ Lock-free algorithms and data structures
- Roadmap

Lock-free algorithms and data structures

Objectives

- Support continuously high indexing rate and concurrent search rate
- Both should be independent, i.e. indexing rate should not decrease when search rate increases and vice versa.

Definitions

- **Pessimistic locking**
    - A thread holds an exclusive lock on a resource, while an action is performed [mutual exclusion]
    - Usually used when conflicts are expected to be likely
- **Optimistic locking**
    - Operations are tried to be performed atomically without holding a lock; conflicts can be detected; retry logic is often used in case of conflicts
    - Usually used when conflicts are expected to be the exception

Definitions

- **Non-blocking algorithm**

  Ensures, that threads competing for shared resources do not have their execution indefinitely postponed by mutual exclusion.

- **Lock-free algorithm**

  A non-blocking algorithm is lock-free if there is guaranteed system-wide progress.

- **Wait-free algorithm**

  A non-blocking algorithm is wait-free, if there is guaranteed per-thread progress.

* Source: Wikipedia

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!

Java Memory Model

- **Program order rule**

  Each action in a thread *happens-before* every action in that thread that comes later in the program order.

- **Volatile variable rule**

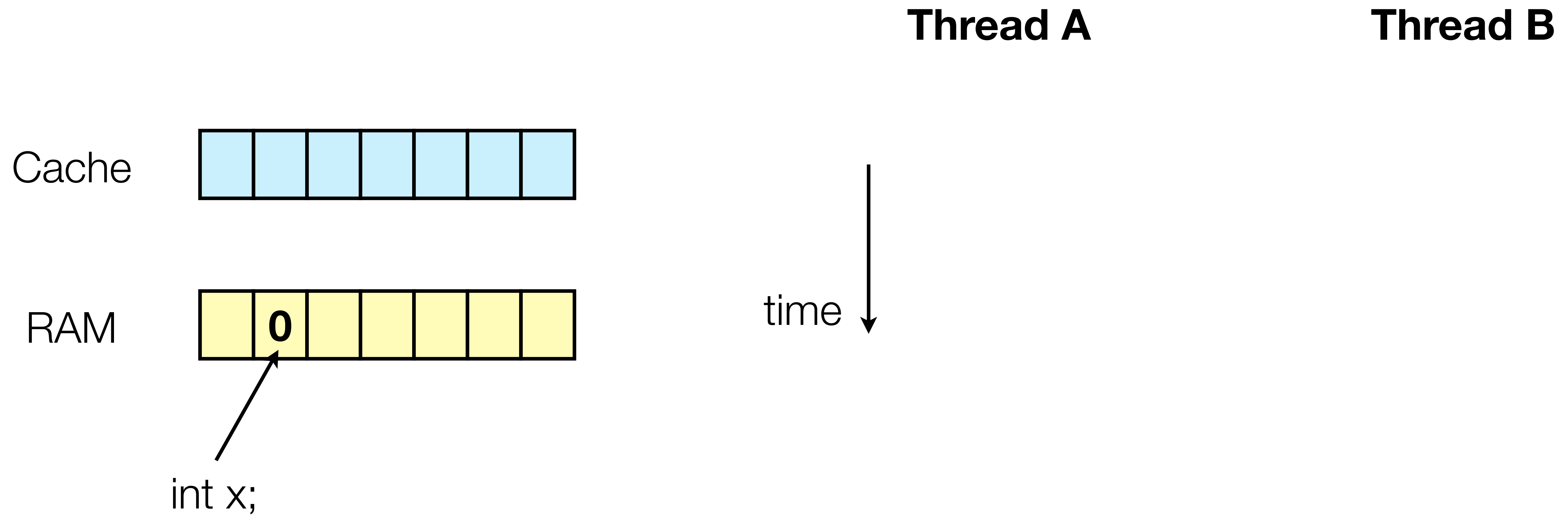
  A write to a volatile field *happens-before* every subsequent read of that same field.

- **Transitivity**

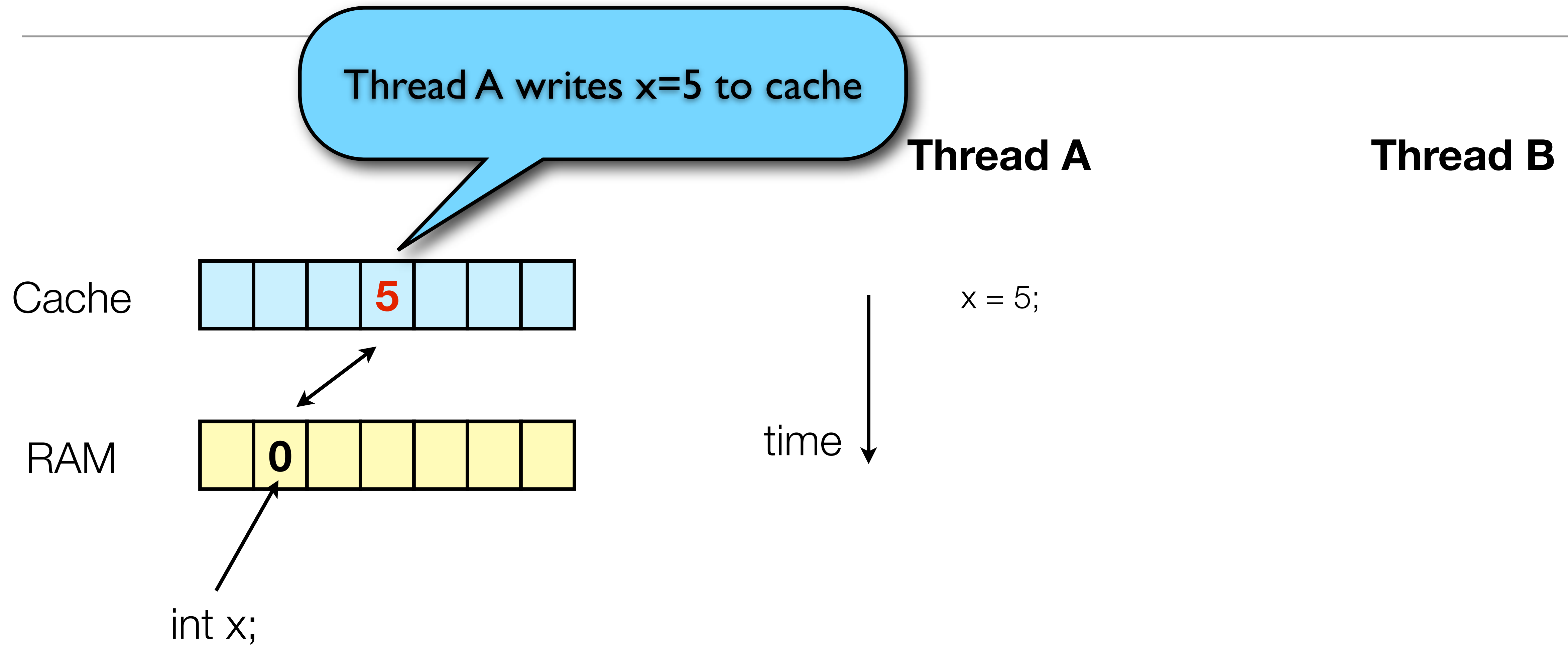
  If A *happens-before* B, and B *happens-before* C, then A *happens-before* C.

* Source: Brian Goetz: Java Concurrency in Practice

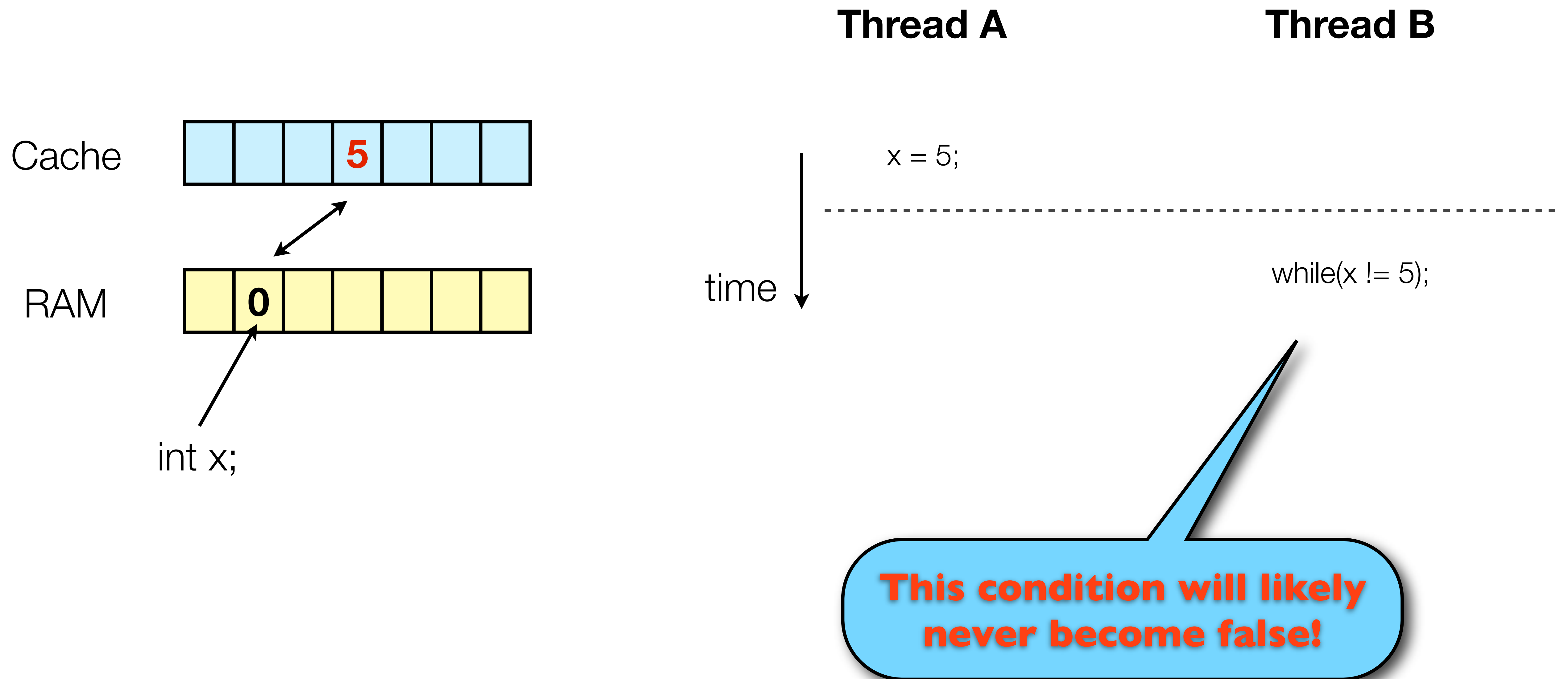
Concurrency



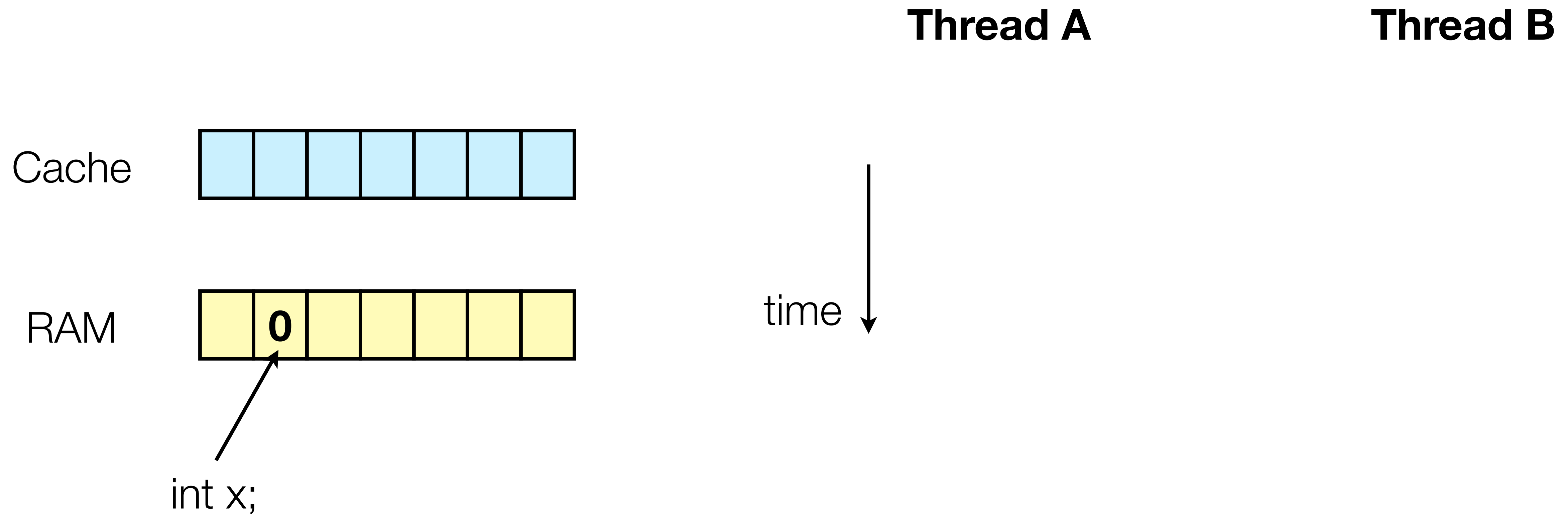
Concurrency



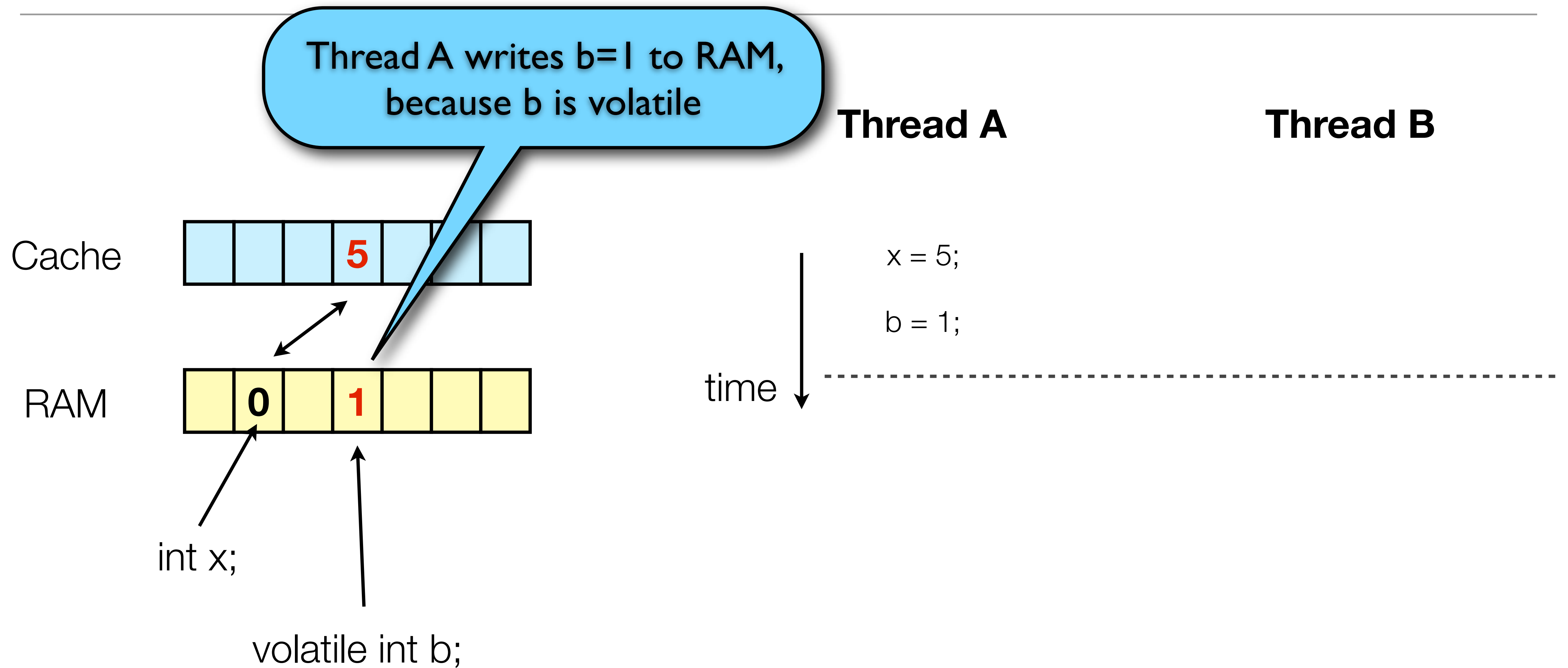
Concurrency



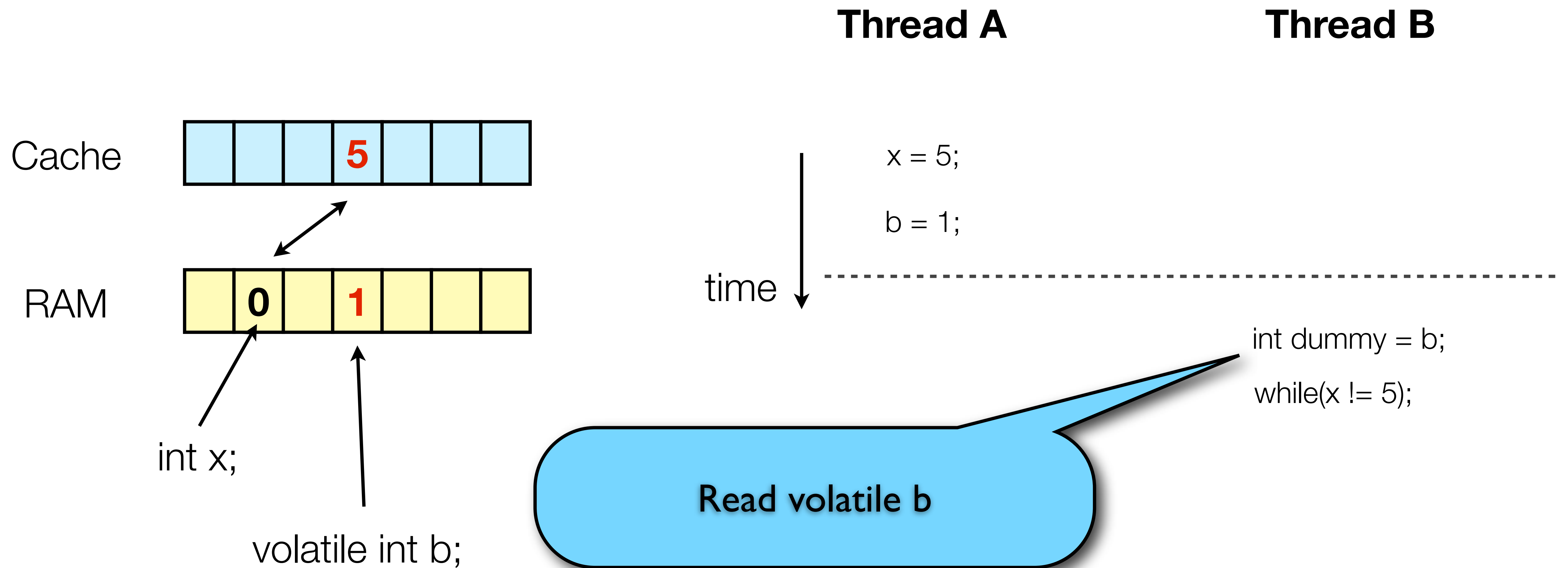
Concurrency



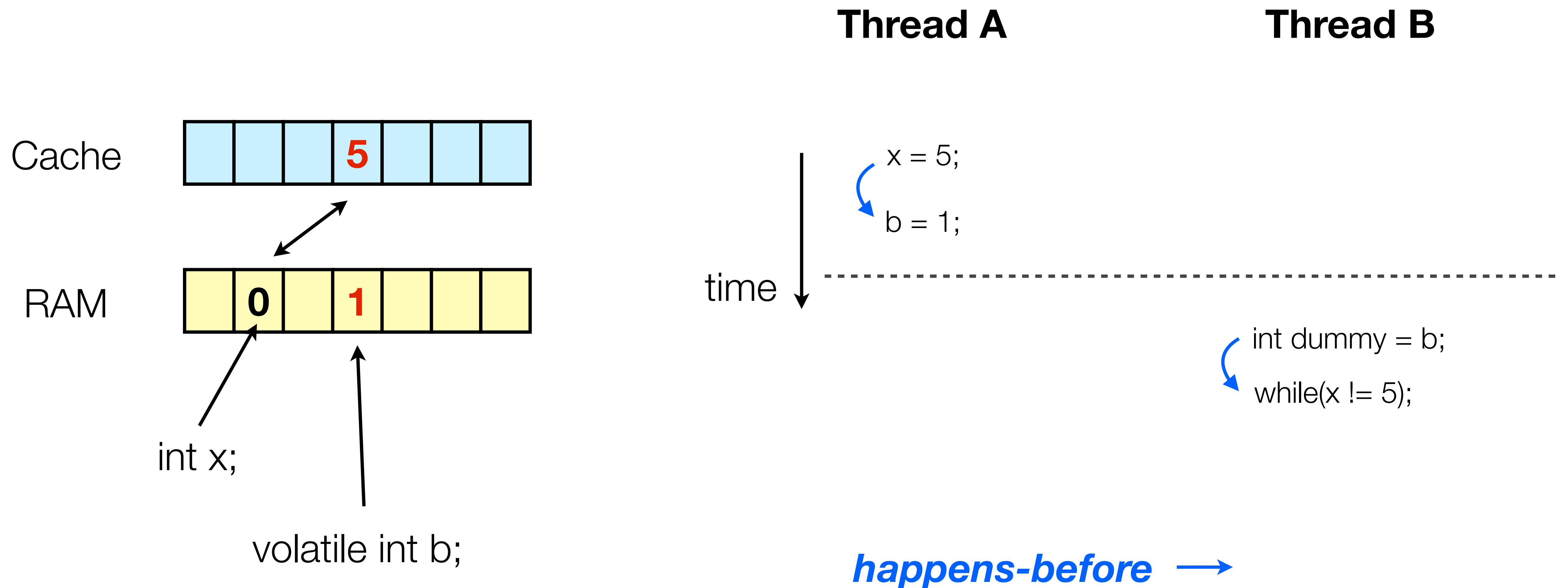
Concurrency



Concurrency

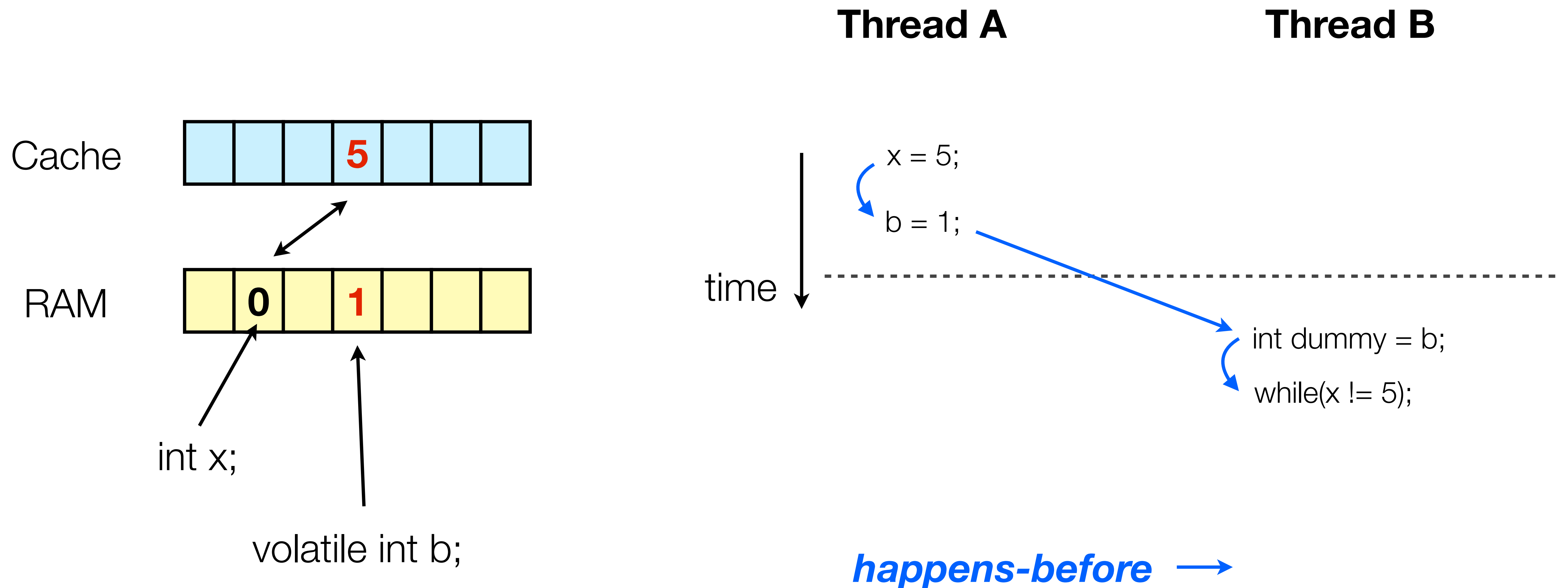


Concurrency



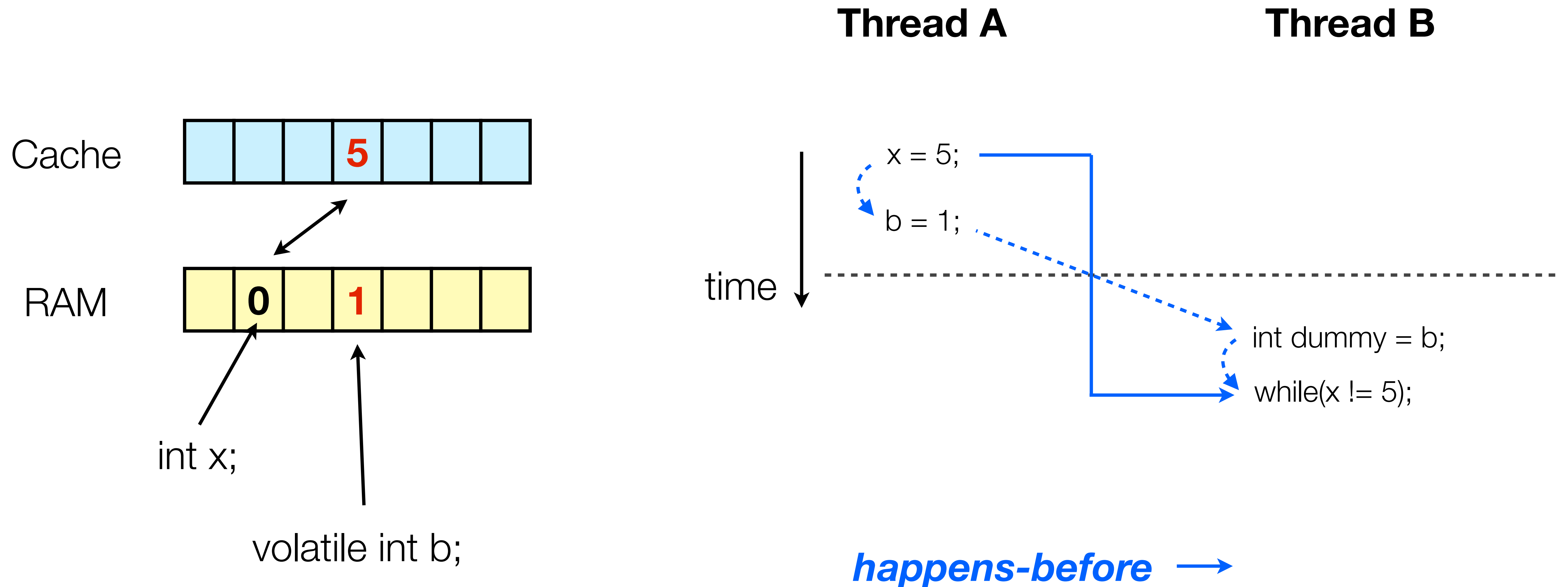
- **Program order rule:** Each action in a thread *happens-before* every action in that thread that comes later in the program order.

Concurrency



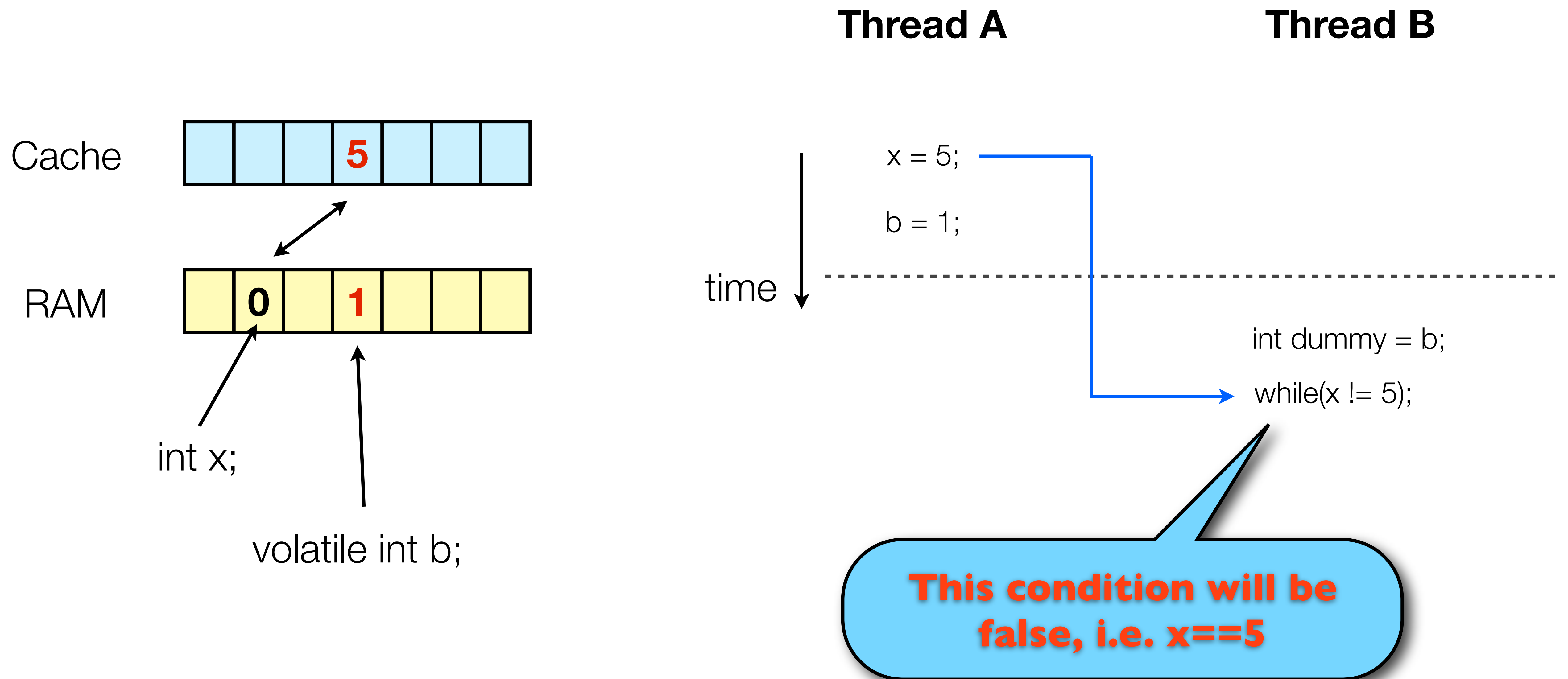
- **Volatile variable rule:** A write to a volatile field *happens-before* every subsequent read of that same field.

Concurrency



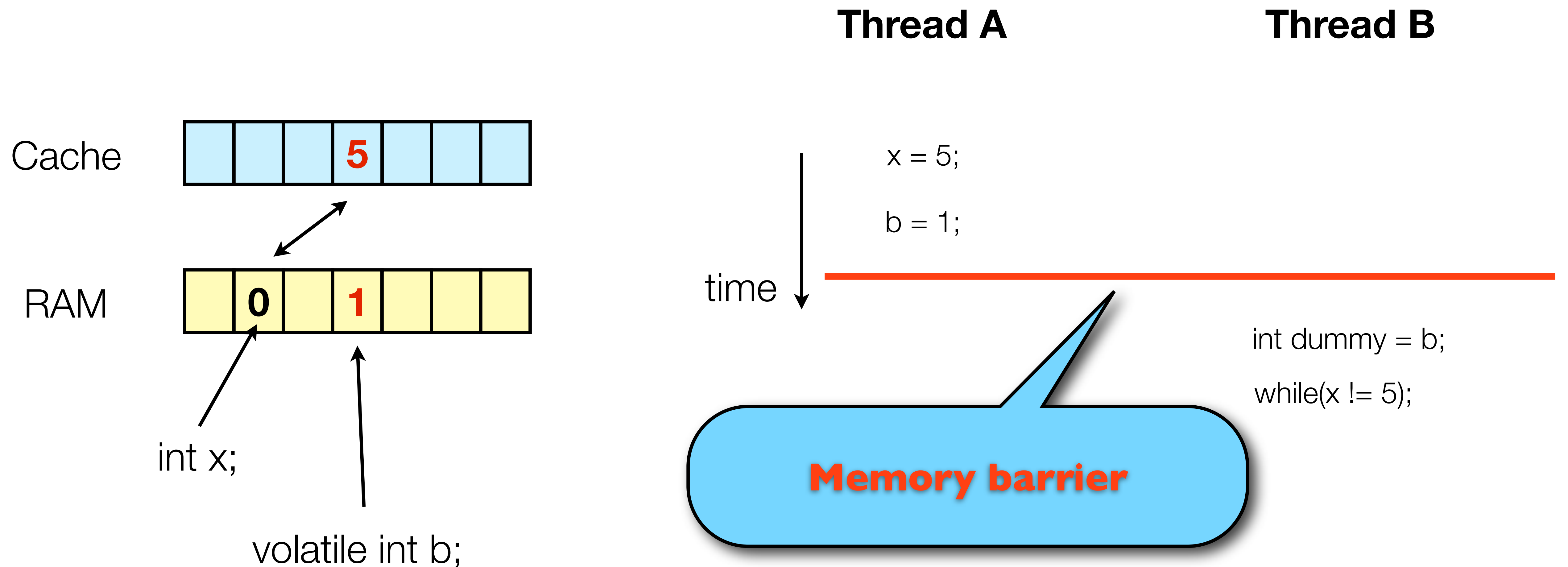
- **Transitivity:** If A *happens-before* B, and B *happens-before* C, then A *happens-before* C.

Concurrency



- **Note:** x itself doesn't have to be volatile. There can be many variables like x, but we need only a single volatile field.

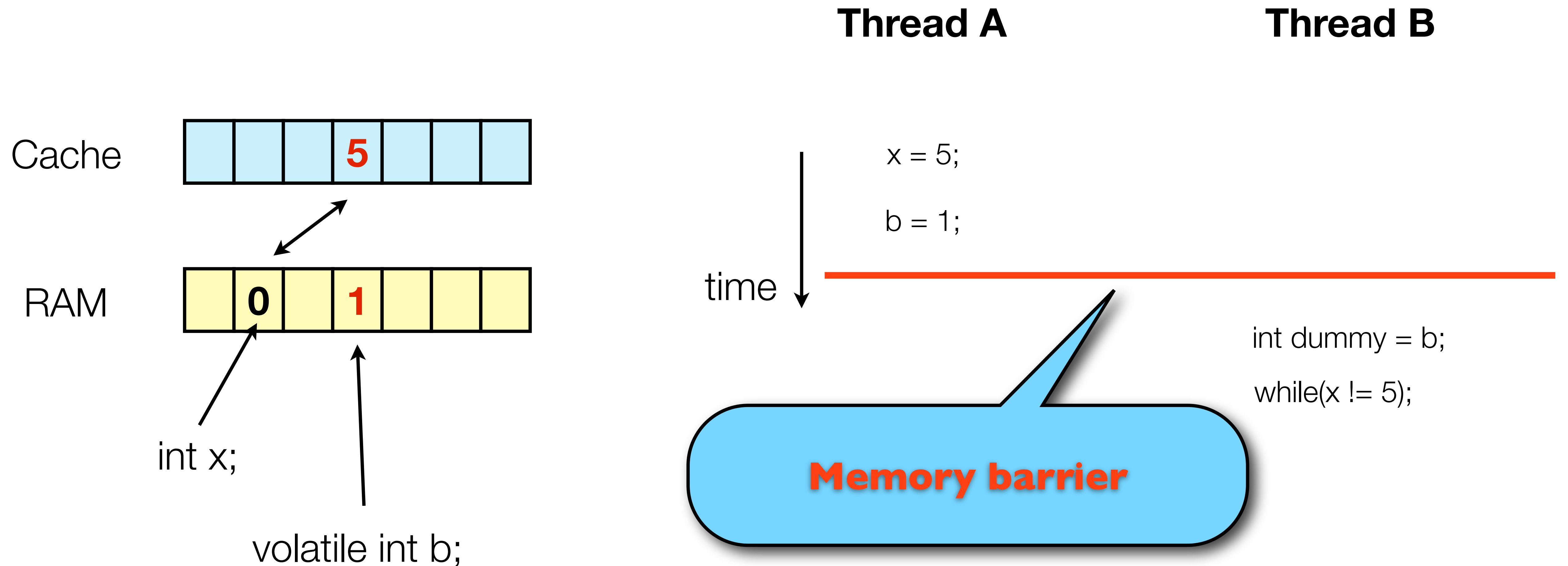
Concurrency



- **Note:** x itself doesn't have to be volatile. There can be many variables like x, but we need only a single volatile field.

Demo

Concurrency



- **Note:** x itself doesn't have to be volatile. There can be many variables like x, but we need only a single volatile field.

Concurrency

IndexWriter **IndexReader**

time

write 100 docs

maxDoc = 100

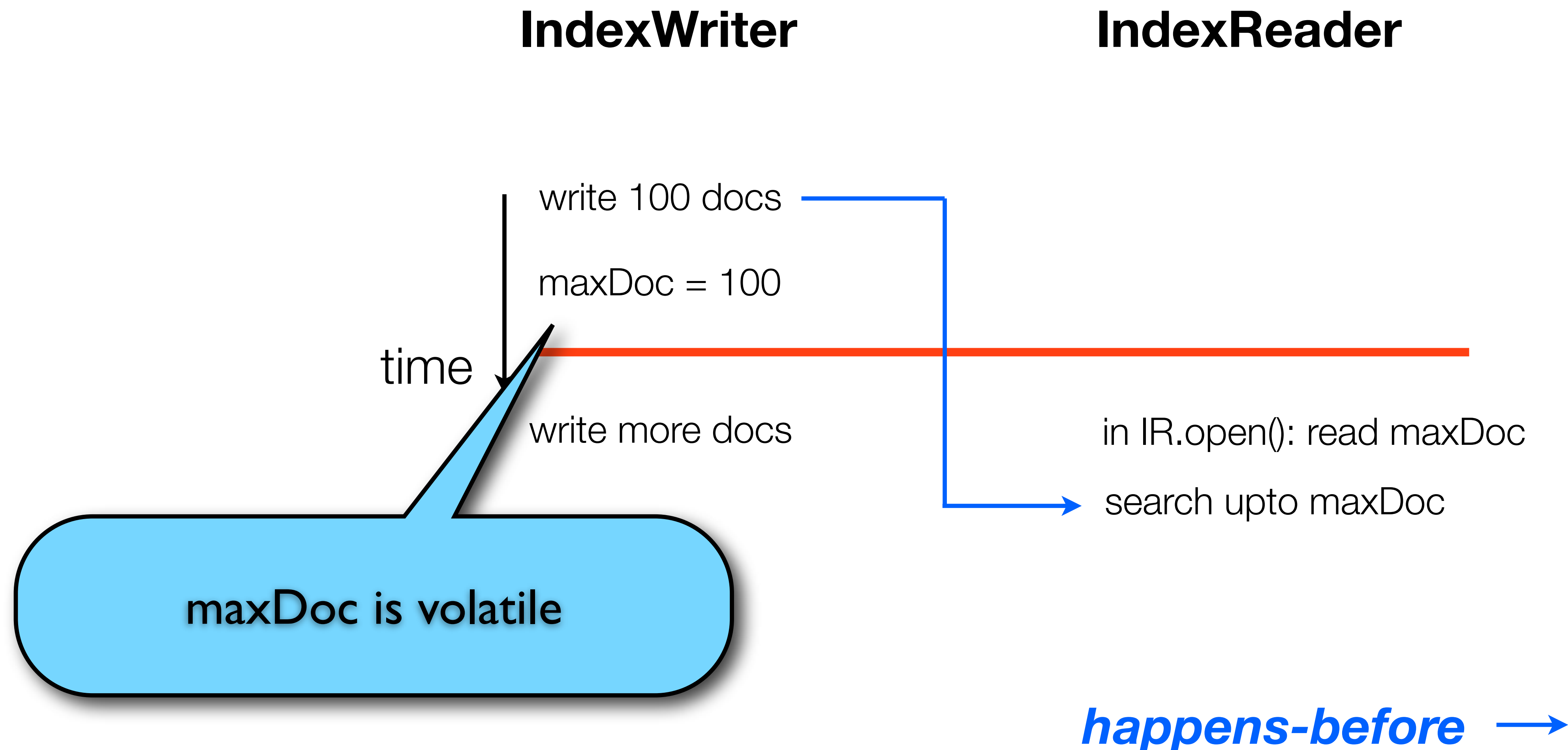
write more docs

in IR.open(): read maxDoc

search upto maxDoc

maxDoc is volatile

Concurrency

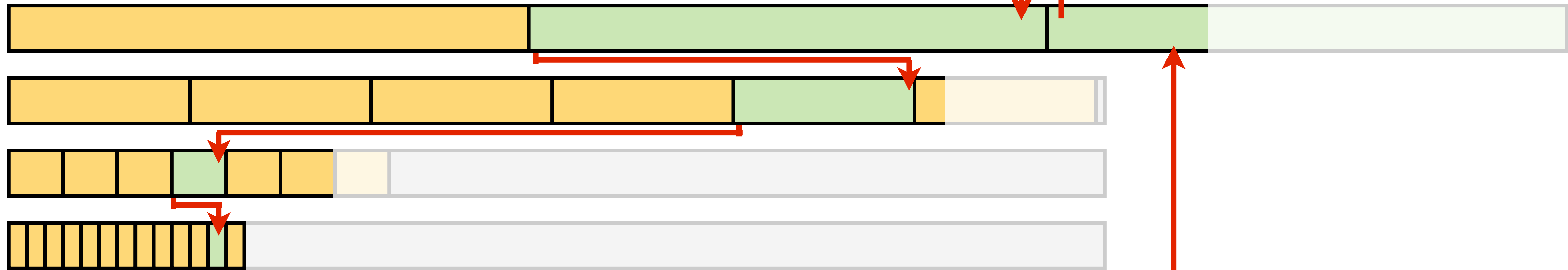


- **Only maxDoc is volatile.** All other fields that IW writes to and IR reads from don't need to be!

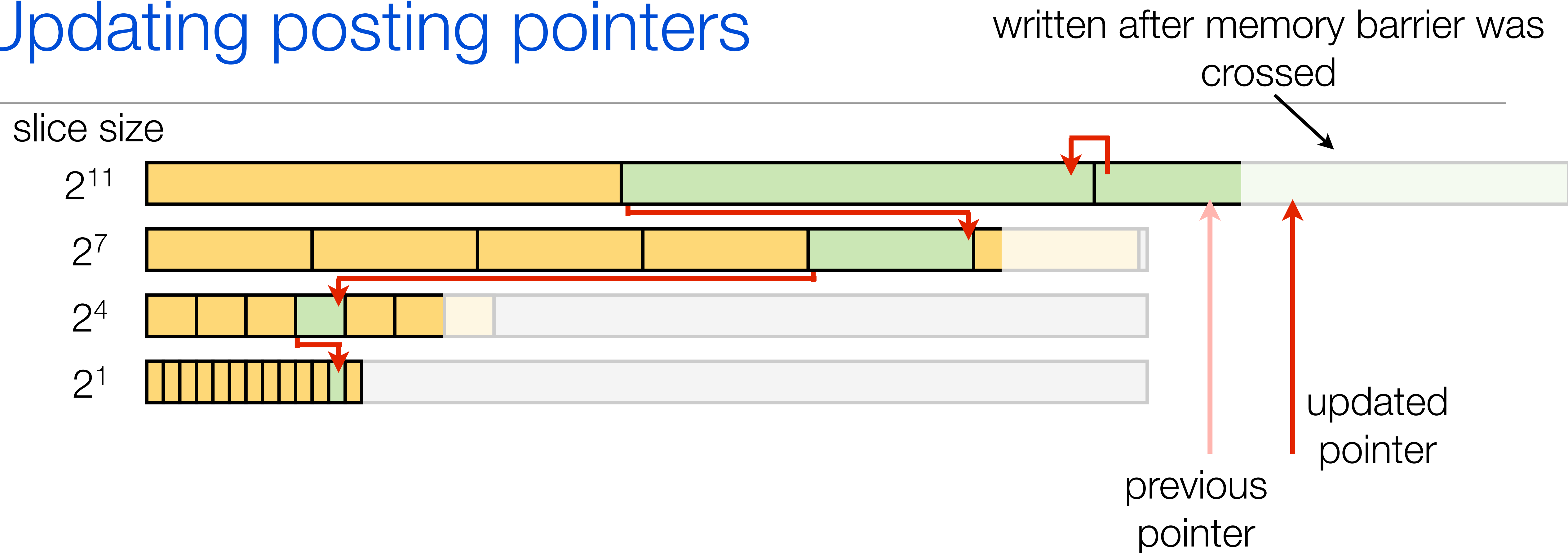
So far...

- Single writer/multi reader approach
- High performance can be achieved by avoiding many volatile/atomic fields
- Safe publication can be achieved by ensuring that memory barriers are crossed
- Opening an in-memory reader is extremely cheap - can be done for every query (at Twitter we open a billion IndexReaders per day!)
- Problem still unsolved: how to update term->postinglist pointers to ensure correct search results?

Updating posting pointers

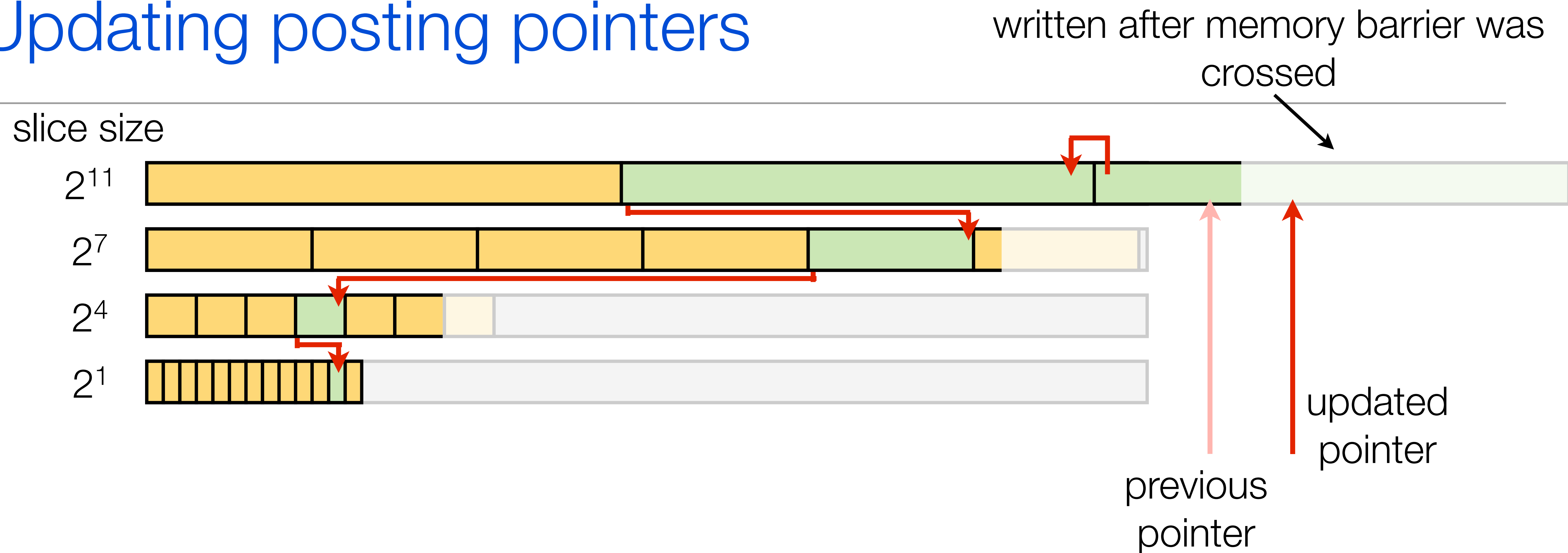


Updating posting pointers



- Safe publication guarantees that everything after a memory barrier is crossed is visible to a thread
- But: a thread might already see a newer state even if the next memory barrier wasn't crossed yet

Updating posting pointers



- This means: a reader might get a postings pointer addressing an int[] block that is not allocated yet!
- We have to handle this case carefully

Updating posting pointers

- Basic idea: keep two most recent pointers
- Writer toggles active pointer whenever memory barrier is crossed
- Reader can detect if a pointer is safe to follow
- Optimistic approach: Reader always considers latest pointer first - only if that one isn't safe it falls back to previous pointer

Wait-free

- Not a single exclusive lock
- Writer thread can always make progress
- Optimistic locking (retry-logic) in a few places for searcher thread
- Retry logic very simple and guaranteed to always make progress - one of the two available posting pointers is always safe to use

Objectives

- Support continuously high indexing rate and concurrent search rate ✓
- Both should be independent, i.e. indexing rate should not decrease when search rate increases and vice versa. ✓

A Billion Queries Per Day

Agenda

- Introduction
- Posting list format and early query termination
- Index memory model
- Lock-free algorithms and data structures
- 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?

A Billion Queries Per Day

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