twitter

230 Million

Tweets per day

2 Billion

Queries per day

< 10 s

Indexing latency

50 ms

Avg. query response time

Earlybird - Realtime Search @twitter

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

Earlybird - Realtime Search @twitter

Agenda

- Introduction
- Search Architecture
- Inverted Index 101
- Memory Model & Concurrency
- Top Tweets

Introduction

Introduction

- Twitter acquired Summize in 2008
- 1st gen search engine based on MySQL

Introduction

- Next gen search engine based on Lucene
- Improves scalability and performance by orders or magnitude
- Open Source

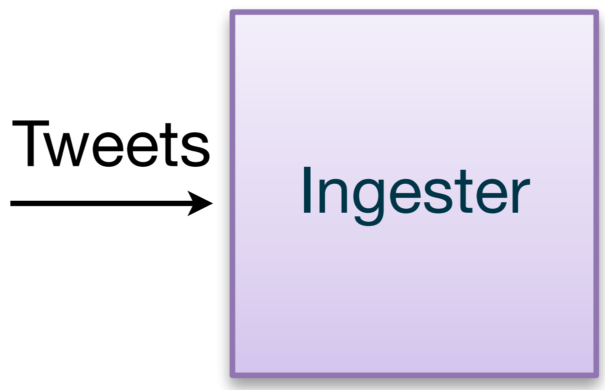
Realtime Search @twitter

Agenda

- Introduction
- Search Architecture
- Inverted Index 101
- Memory Model & Concurrency
- Top Tweets

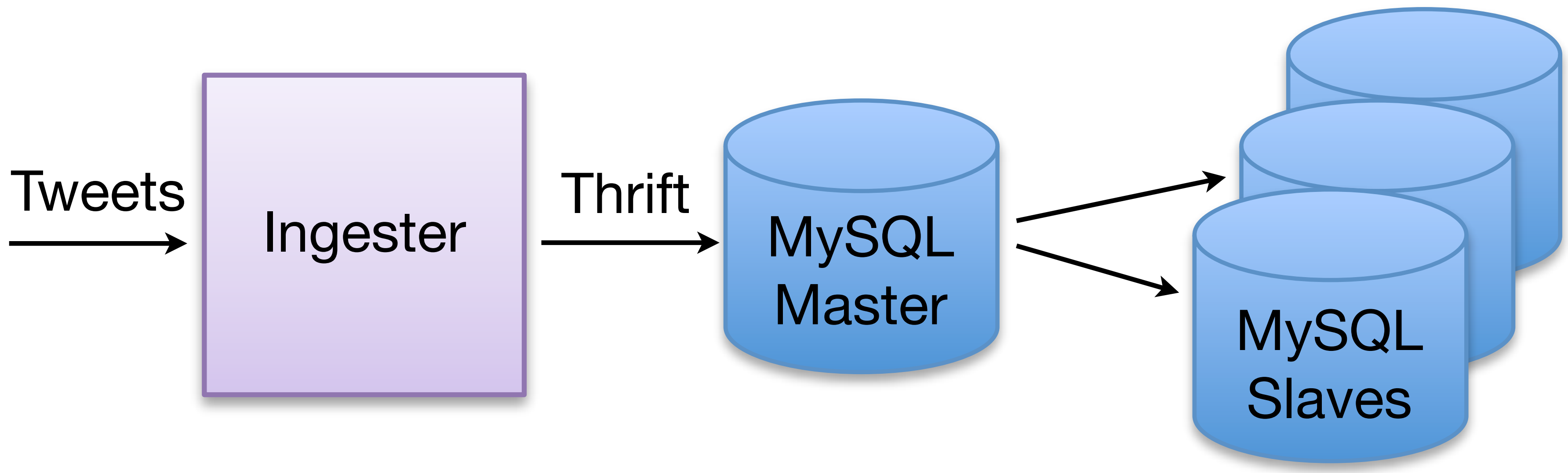
Search Architecture

Search Architecture



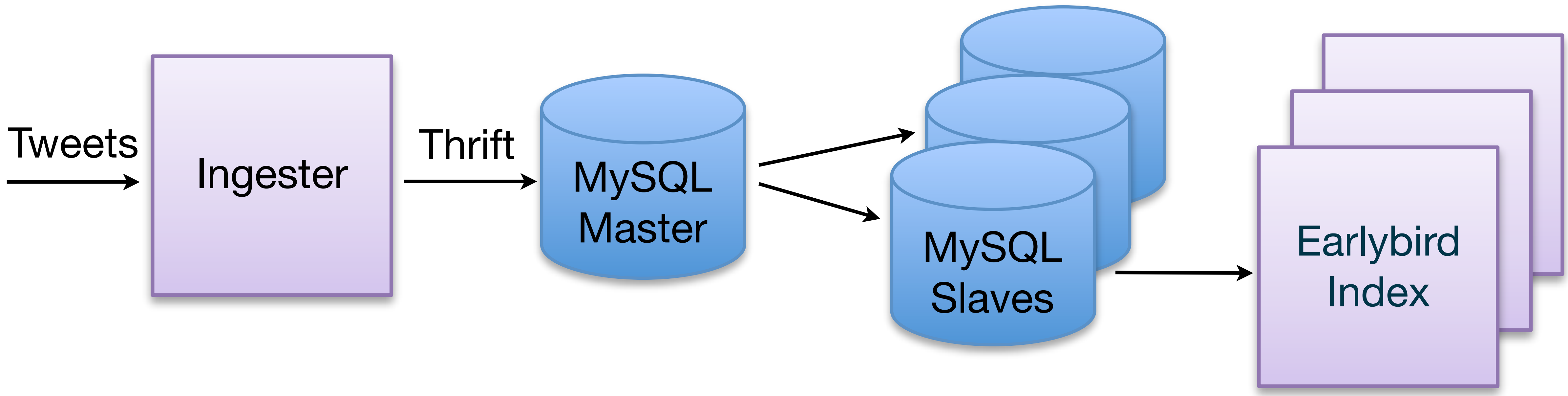
- Ingester pre-processes Tweets for search
- Geo-coding, URL expansion, tokenization, etc.

Search Architecture



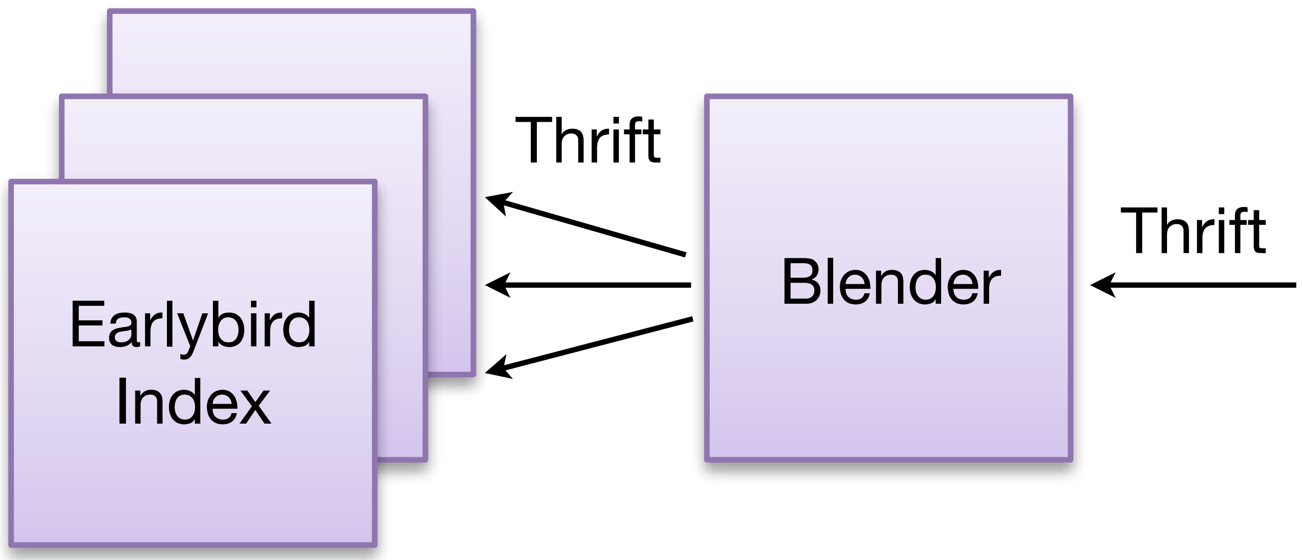
- Tweets are serialized to MySQL in Thrift format

Earlybird



- Earlybird reads from MySQL slaves
- Builds an in-memory inverted index in real time

Blender



- Blender is our Thrift service aggregator
- Queries multiple Earlybirds, merges results

Realtime Search @twitter

Agenda

- Introduction
- Search Architecture
- Inverted Index 101
- Memory Model & Concurrency
- Top Tweets

Inverted Index 101

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

Posting list encoding

Doc IDs to encode: 5, 15, 9000, 9002, 100000, 100090

Posting list encoding

Doc IDs to encode: 5, 15, 9000, 9002, 100000, 100090

Delta encoding:

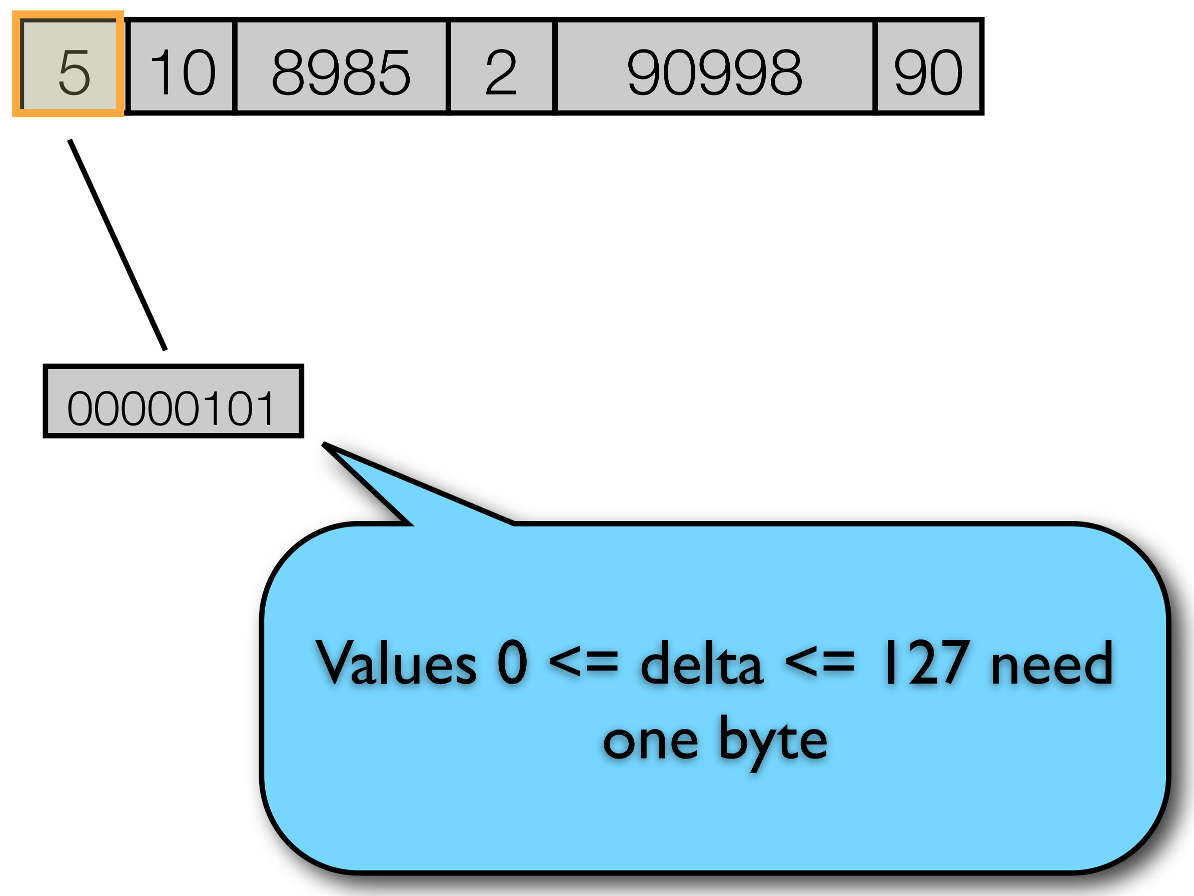
5	10	8985	2	90998	90

Posting list encoding

Doc IDs to encode: 5, 15, 9000, 9002, 100000, 100090

Delta encoding:

VInt compression:



Posting list encoding

Doc IDs to encode: 5, 15, 9000, 9002, 100000, 100090

Delta encoding:

5	10	8985	2	90998	90

VInt compression:

11000110	00011001

Values 128 <= delta <= 16384
need two bytes

Posting list encoding

Doc IDs to encode: 5, 15, 9000, 9002, 100000, 100090

Delta encoding: | 5 | 10 | 8985 | 2 | 90998 | 90 |

VInt compression: | **1**1000110 | **0**0011001 |

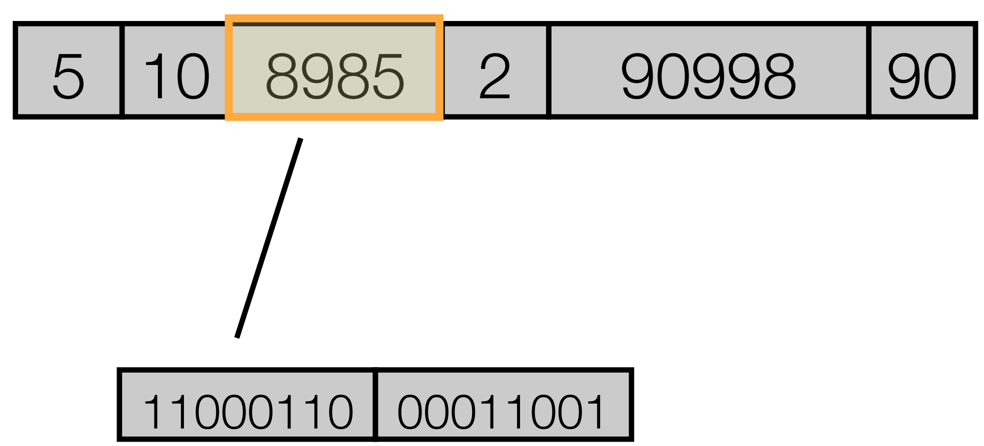
First bit indicates whether next byte belongs to the same value

Posting list encoding

Doc IDs to encode: 5, 15, 9000, 9002, 100000, 100090

Delta encoding:

VInt compression:



- Variable number of bytes - a VInt-encoded posting can not be written as a primitive Java type; therefore it can not be written atomically

Posting list encoding

Doc IDs to encode: 5, 15, 9000, 9002, 100000, 100090

Delta encoding:

5	10	8985	2	90998	90

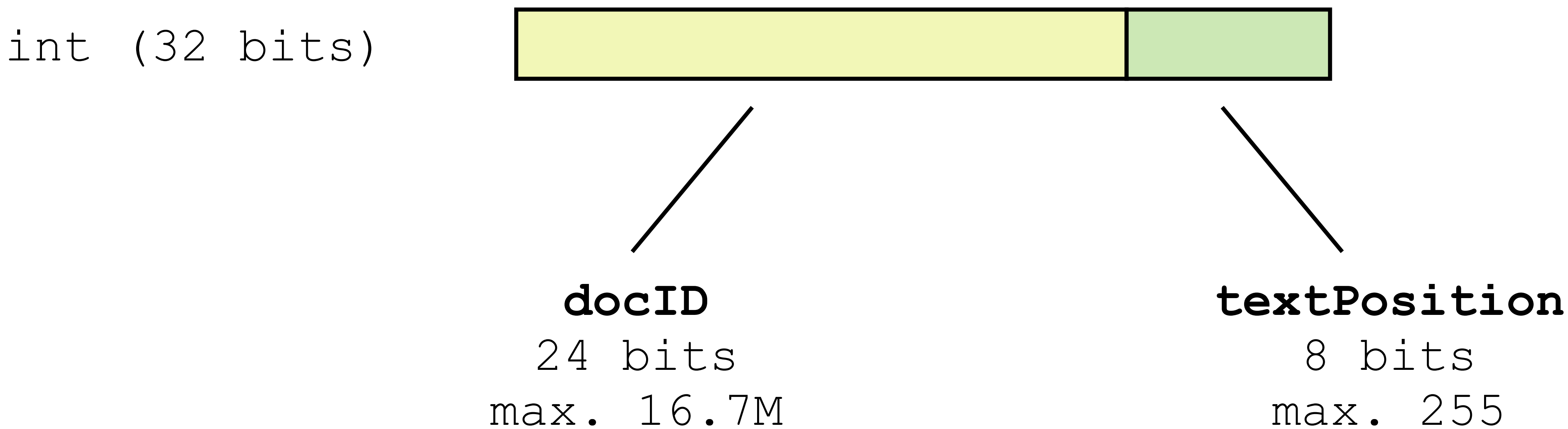
Read direction

- Each posting depends on previous one; decoding only possible in old-to-new direction
- With recency ranking (new-to-old) no early termination is possible

Posting list encoding

- By default 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?

Posting list encoding in Earlybird

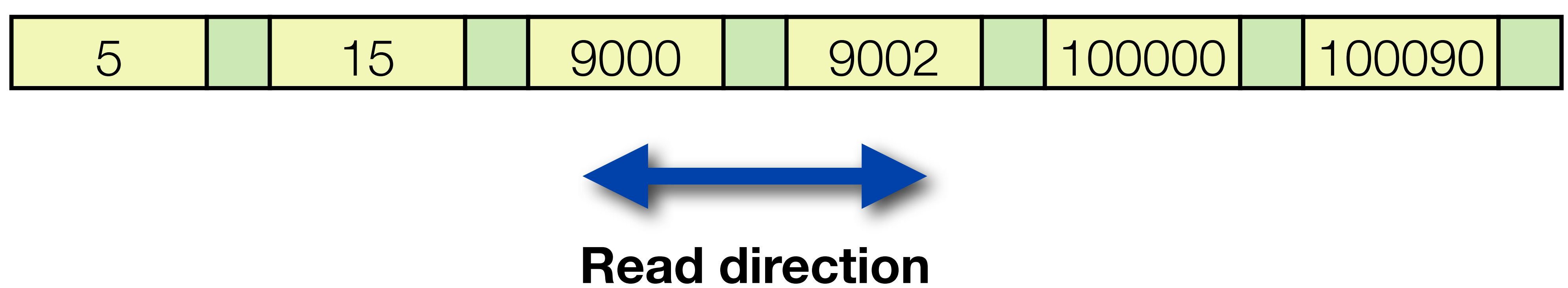


- Tweet text can only have 140 chars
- Decoding speed significantly improved compared to delta and VInt decoding (early experiments suggest 5x improvement compared to vanilla Lucene with FSDirectory)

Posting list encoding in Earlybird

Doc IDs to encode: 5, 15, 9000, 9002, 100000, 100090

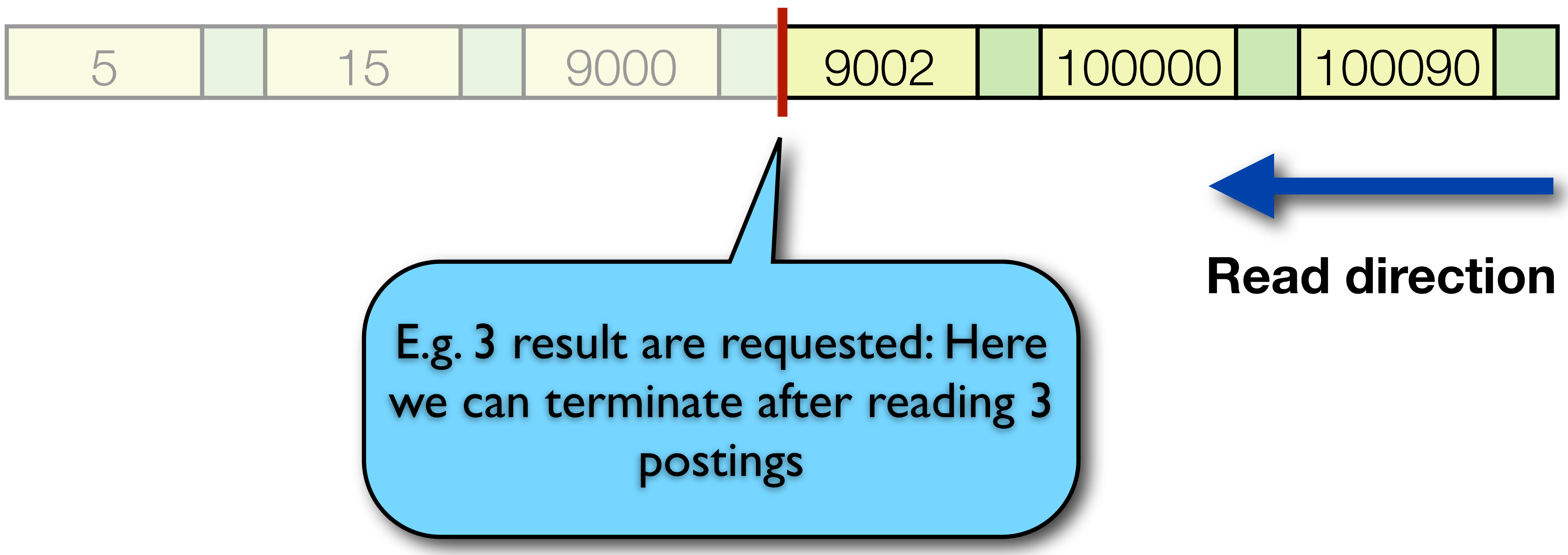
Earlybird encoding:



Early query termination

Doc IDs to encode: 5, 15, 9000, 9002, 100000, 100090

Earlybird encoding:



Posting list encoding - Summary

- 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; compensated by compression of complete segments
- Max. segment size: 2^24 = 16.7M tweets

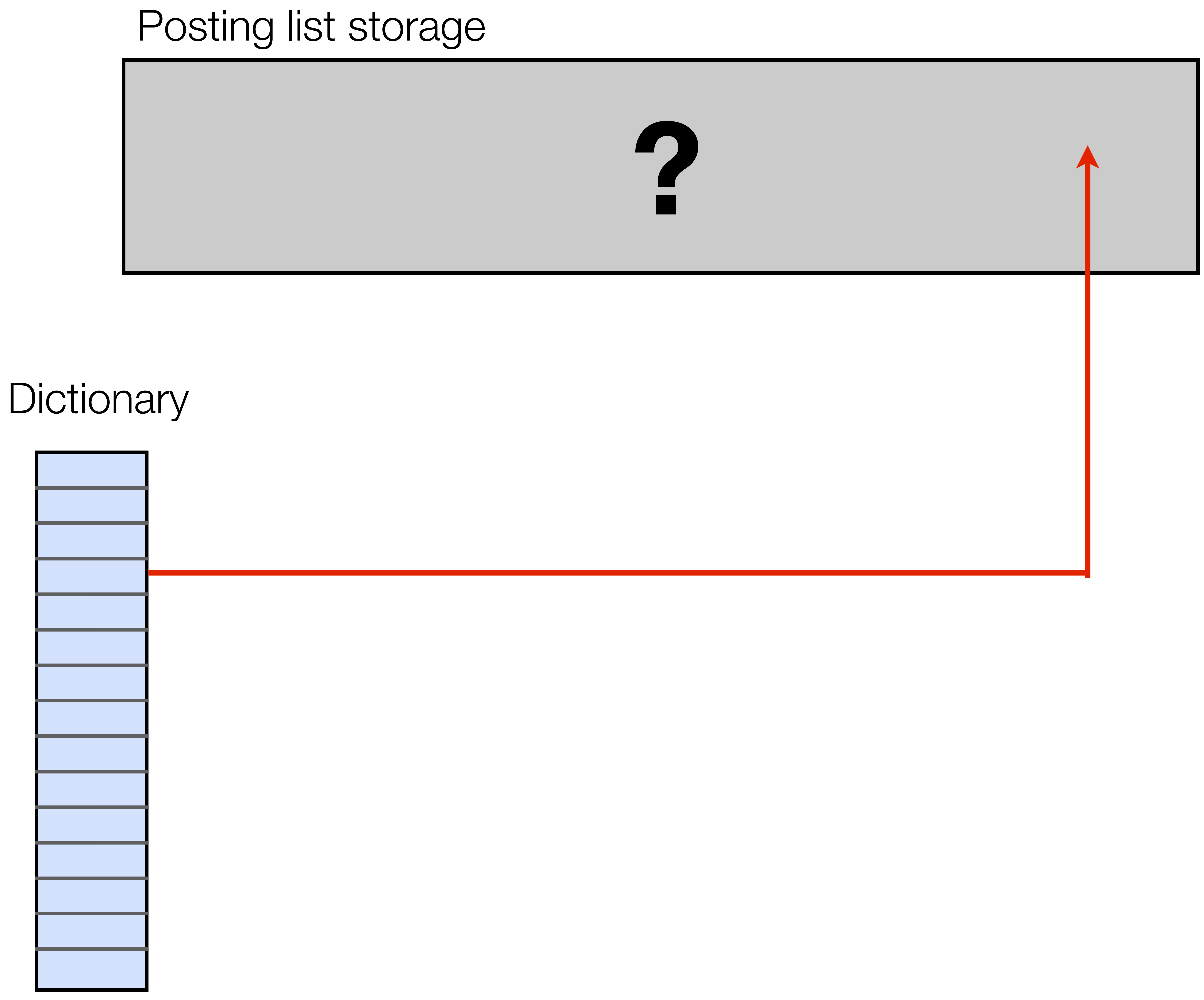
Realtime Search @twitter

Agenda

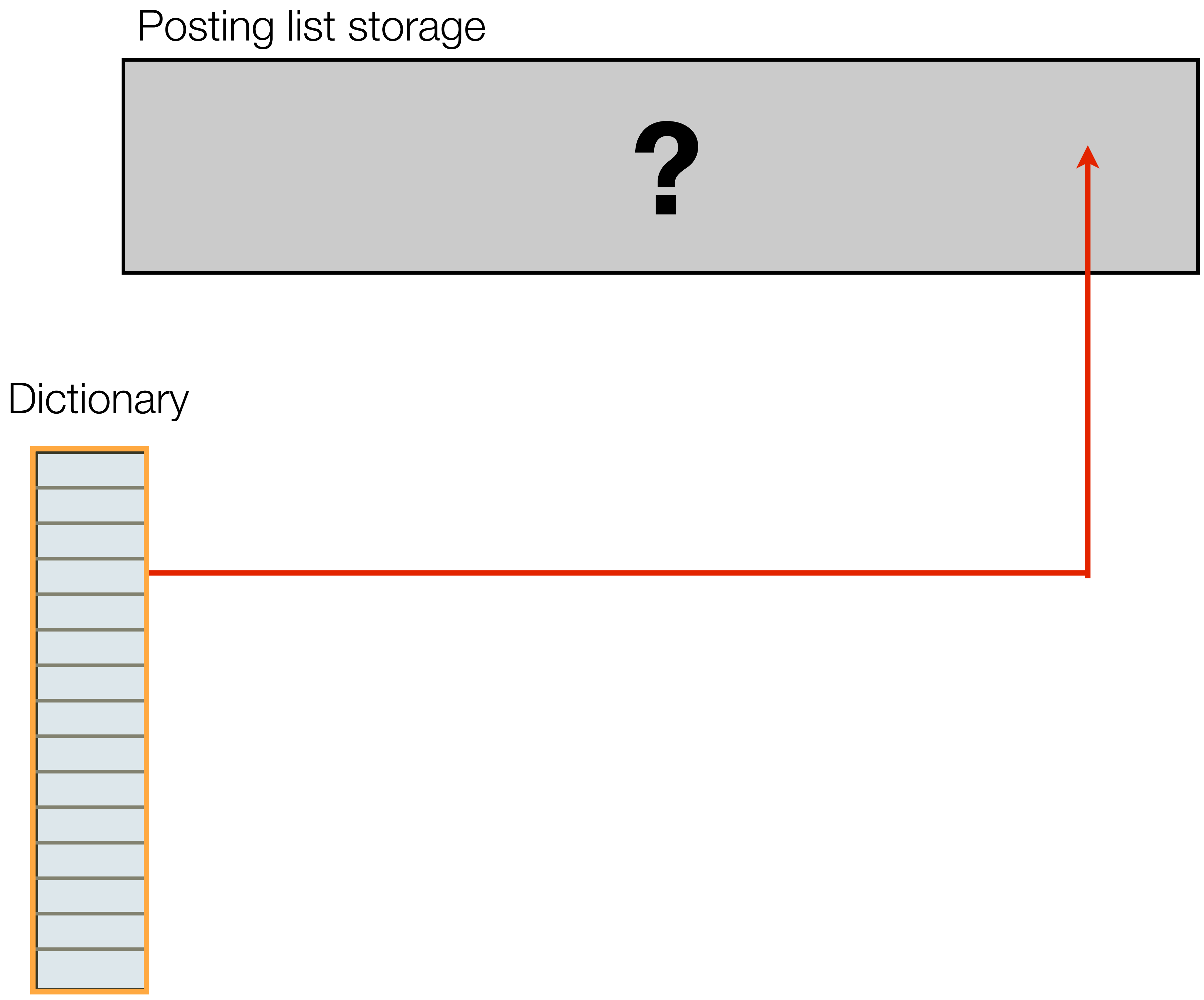
- Introduction
- Search Architecture
- Inverted Index 101
- ‣ Memory Model & Concurrency
- Top Tweets

Memory Model & Concurrency

Inverted index components



Inverted index components



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.

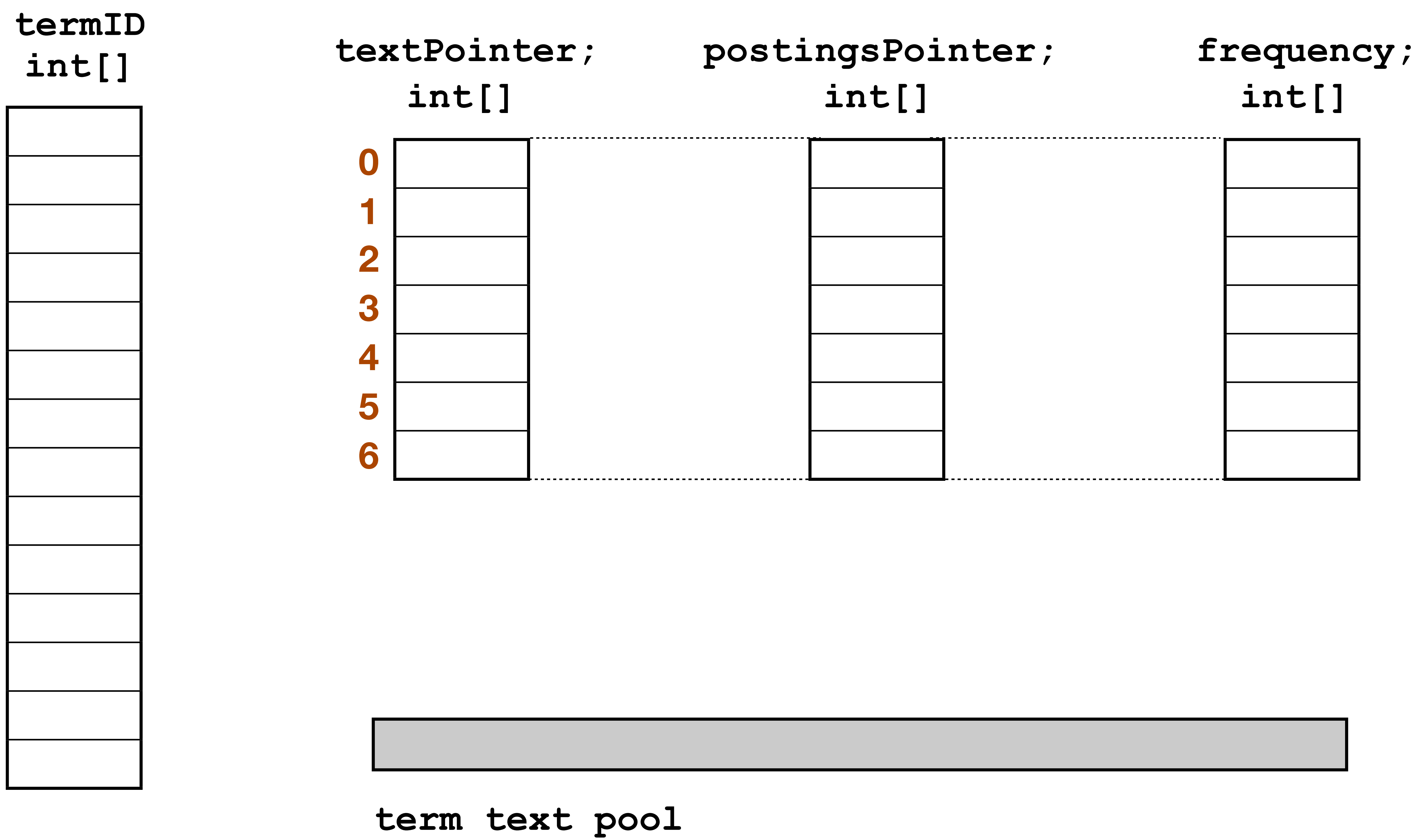
Table with 6 documents

term	freq	
and	[illegible]	<6>
big	[illegible]	[illegible]
dark	[illegible]	[illegible]
did	[illegible]	[illegible]
gown	[illegible]	[illegible]
had	[illegible]	[illegible]
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>

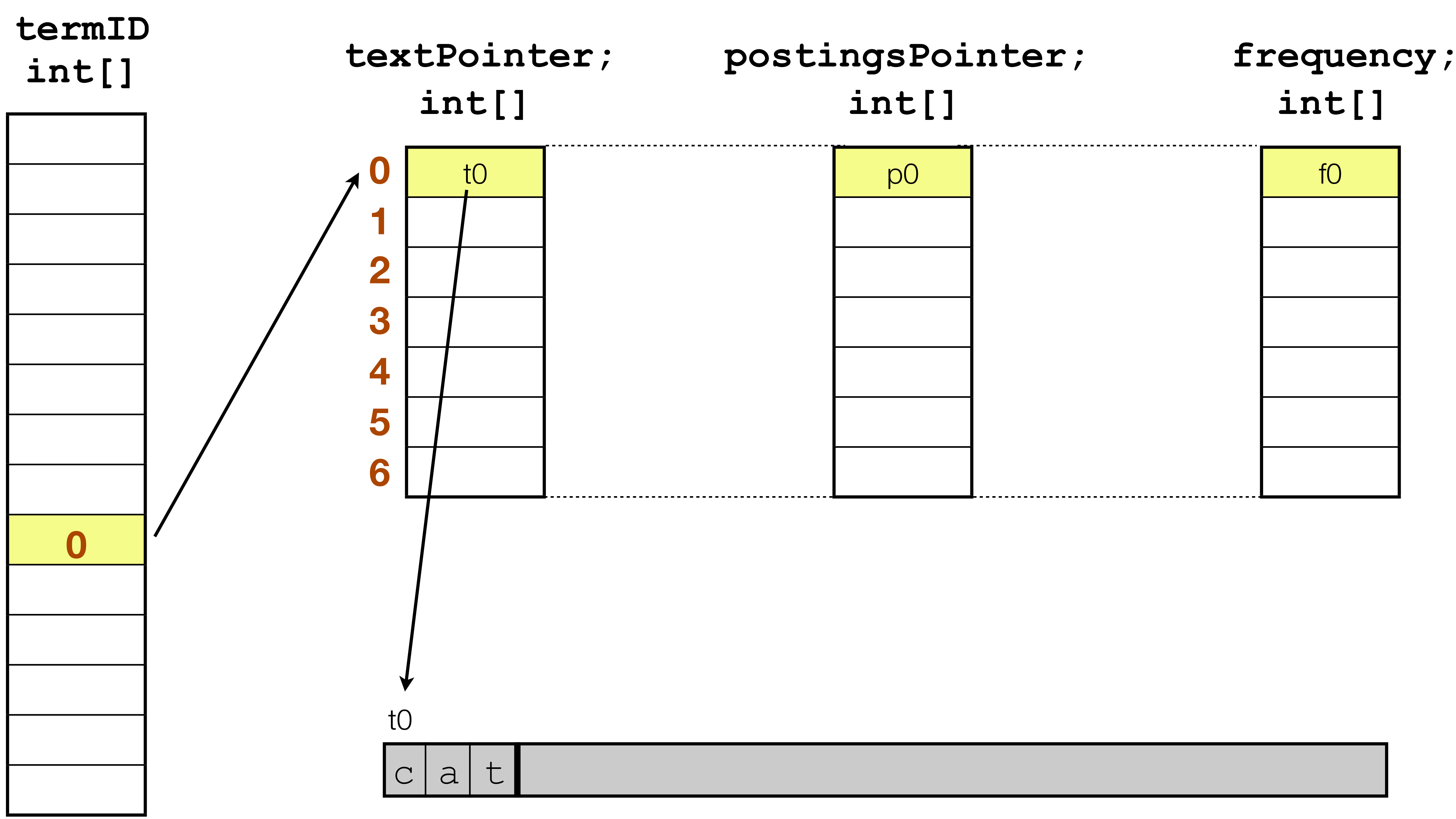
Per term we store different kinds of metadata: text pointer, frequency, postings pointer, etc.

Dictionary and posting lists

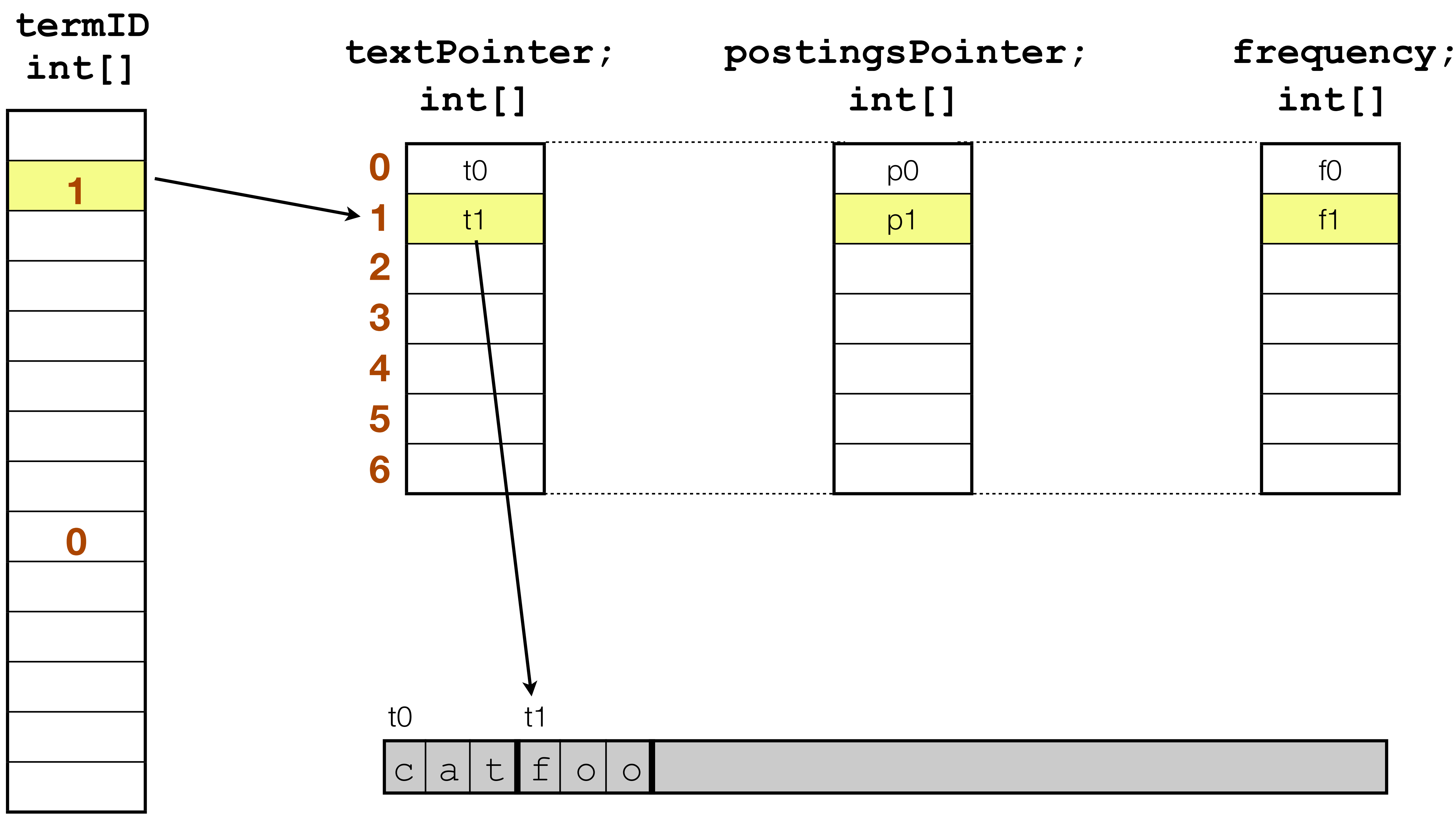
Term dictionary



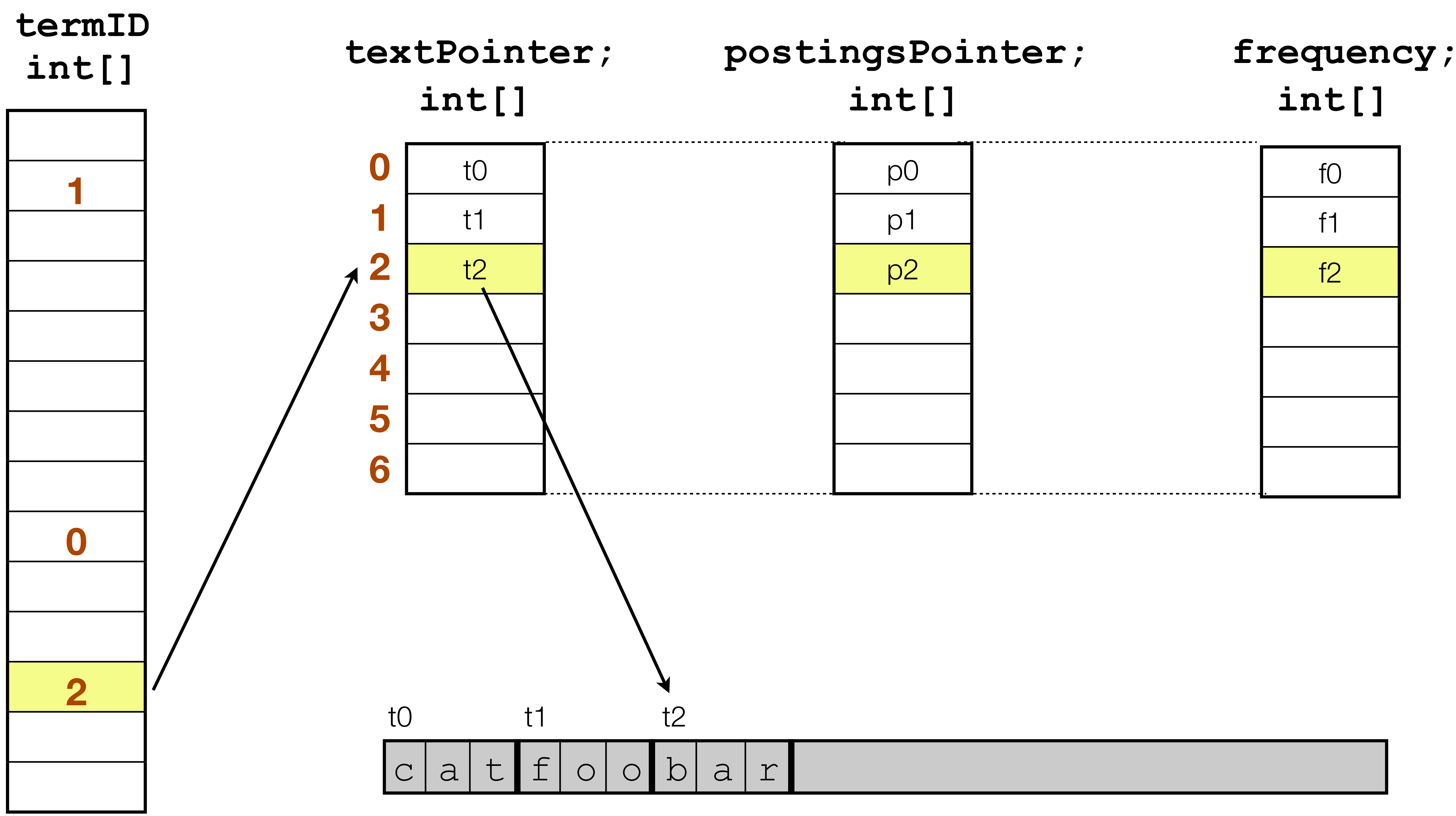
Term dictionary



Term dictionary



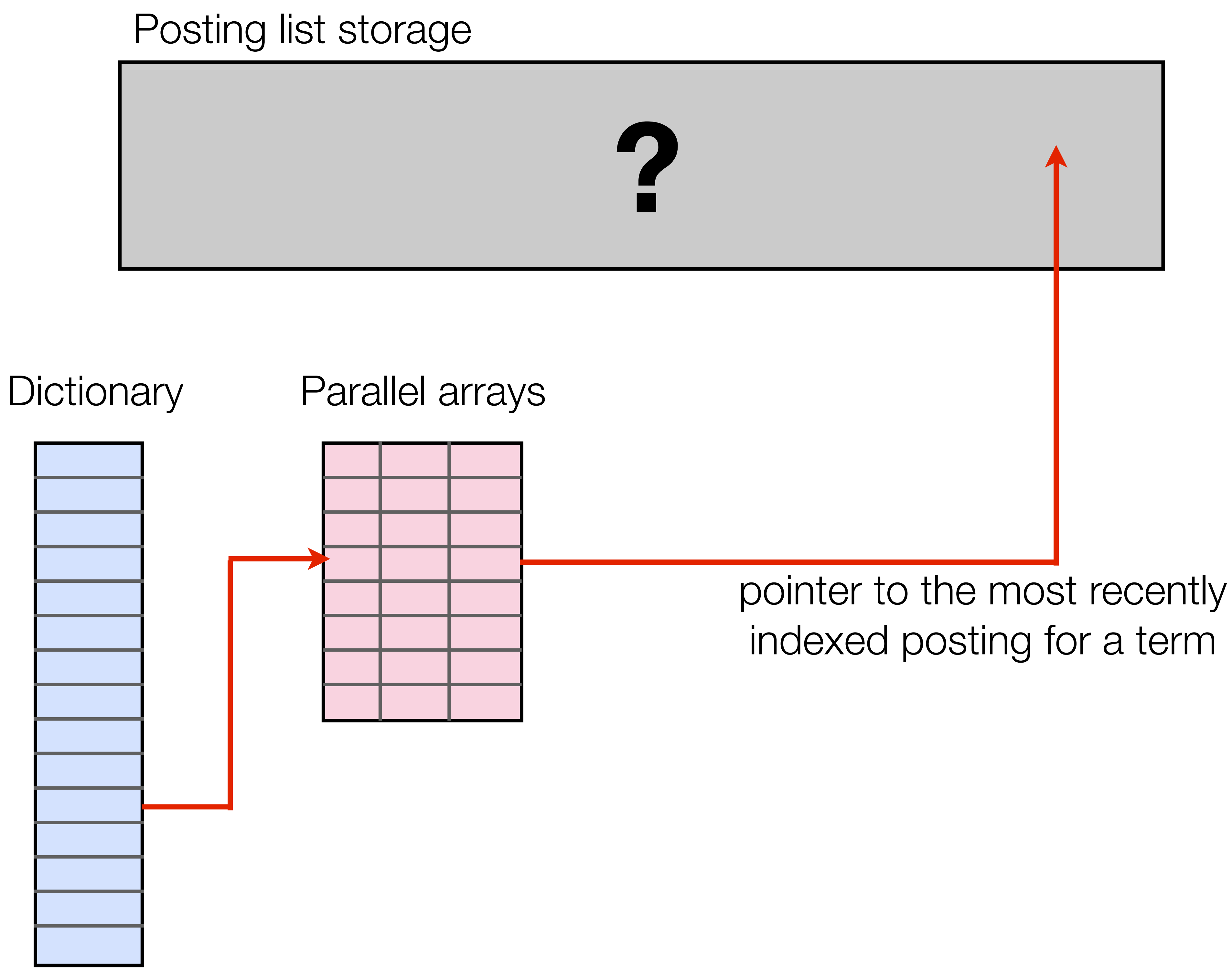
Term dictionary



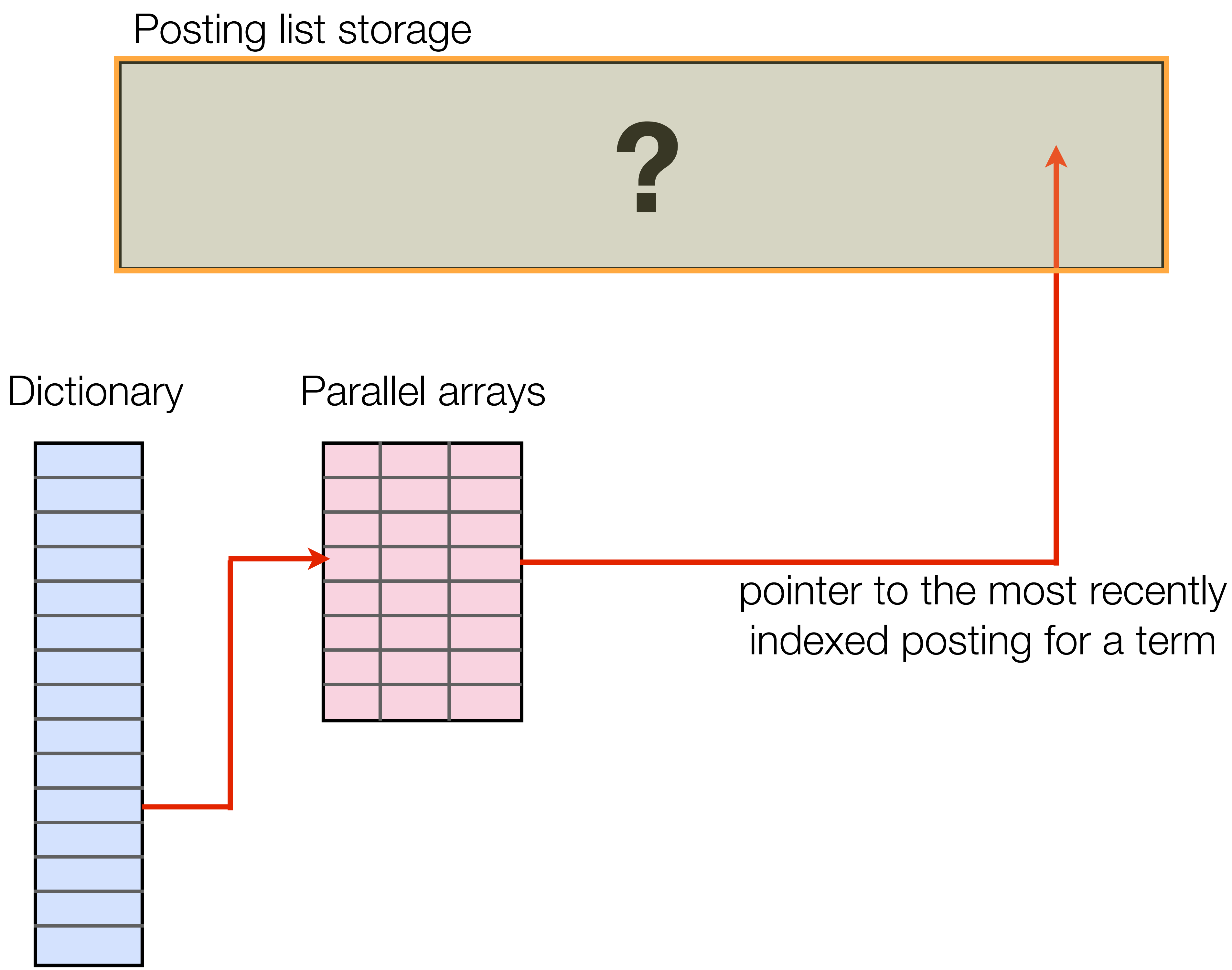
Term dictionary

- Number of objects << number of terms
- O(1) lookups
- Easy to store more term metadata by adding additional parallel arrays

Inverted index components



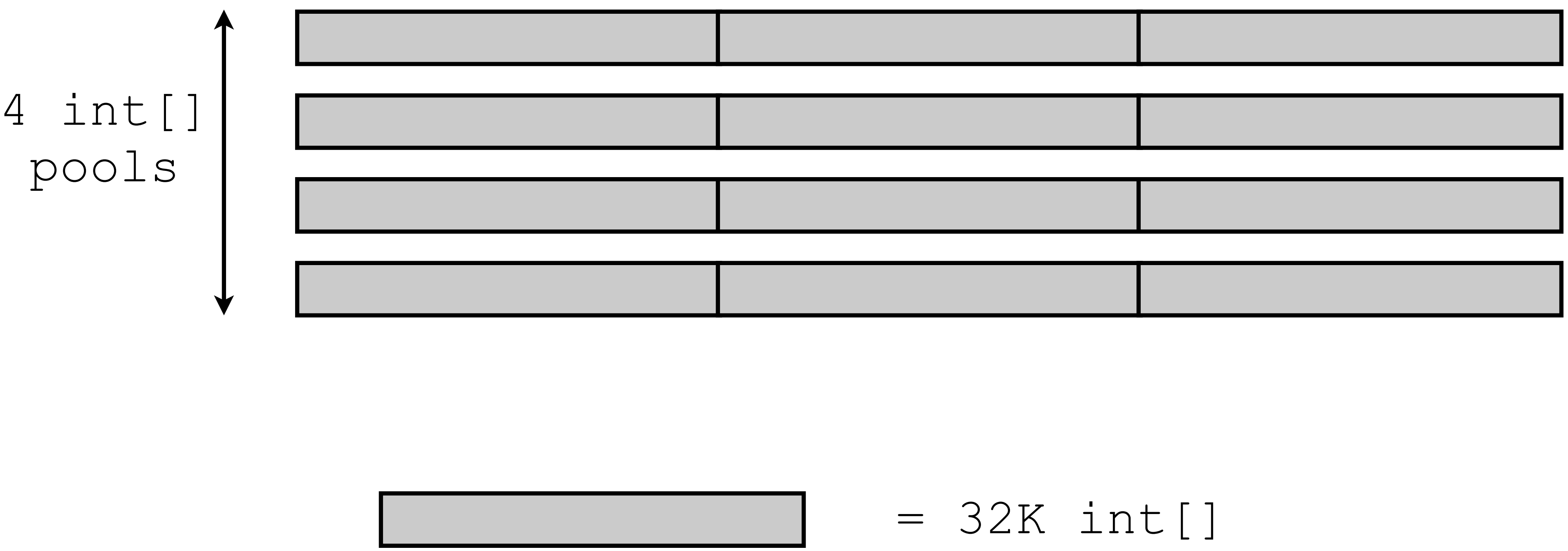
Inverted index components



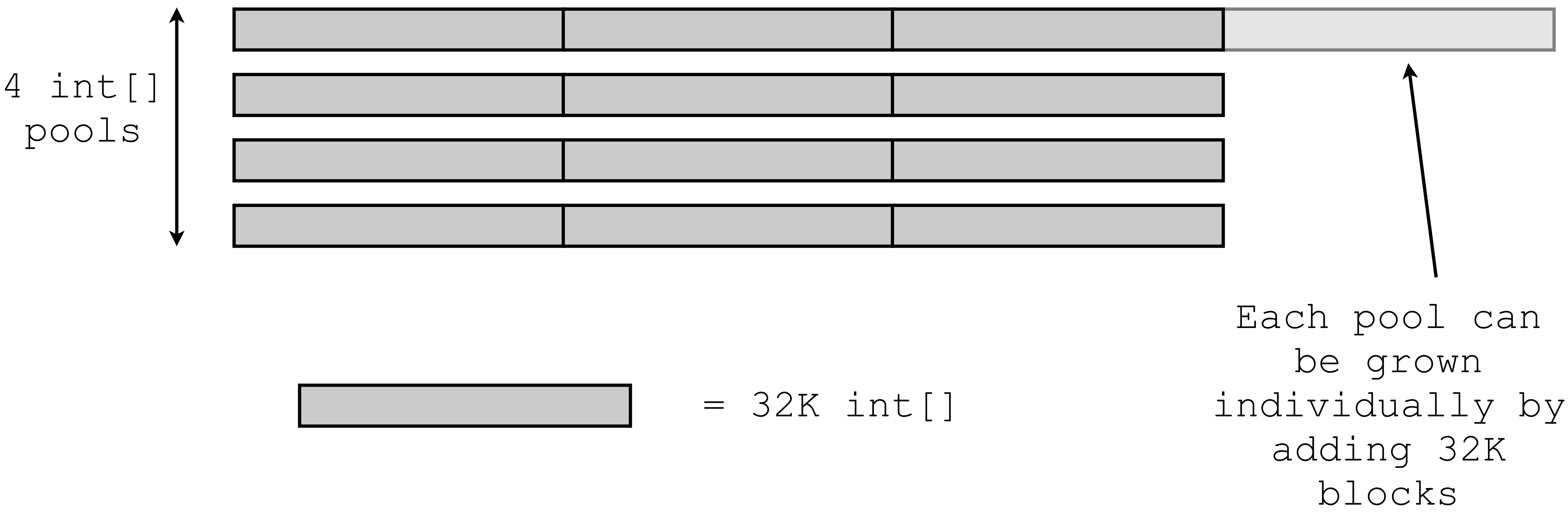
Posting lists storage - Objectives

- 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

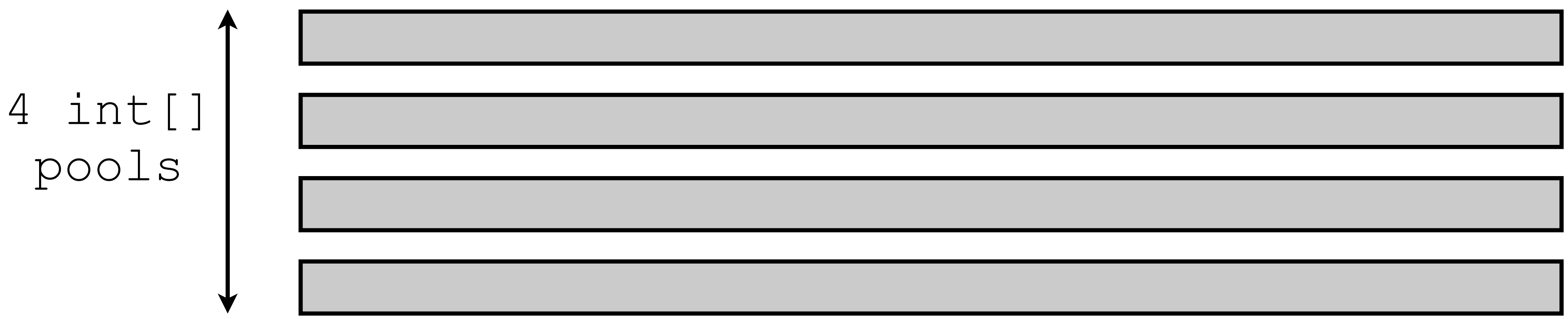
Memory management



Memory management



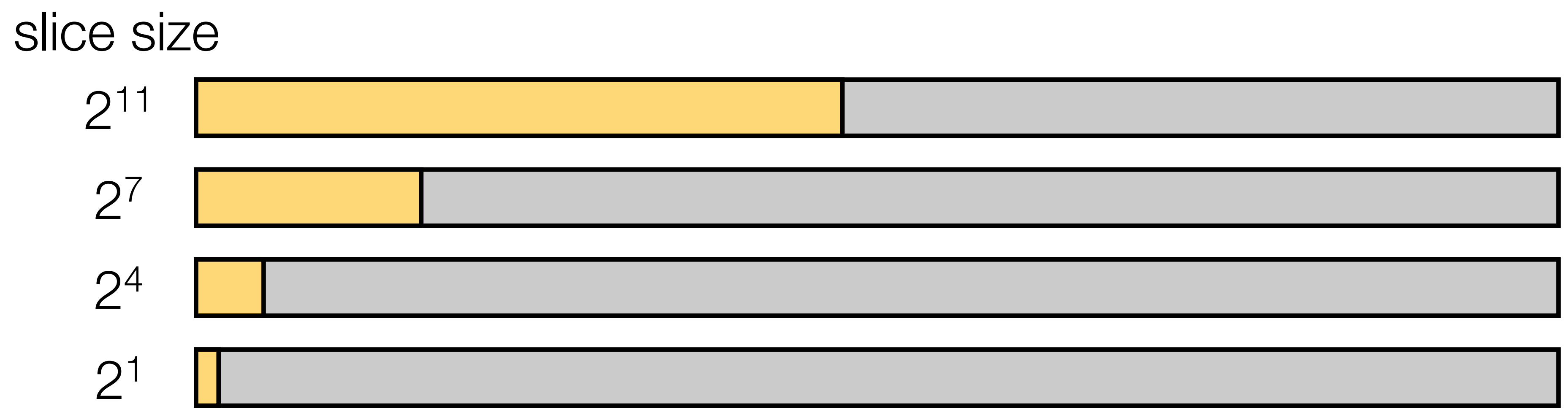
Memory management



- 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)

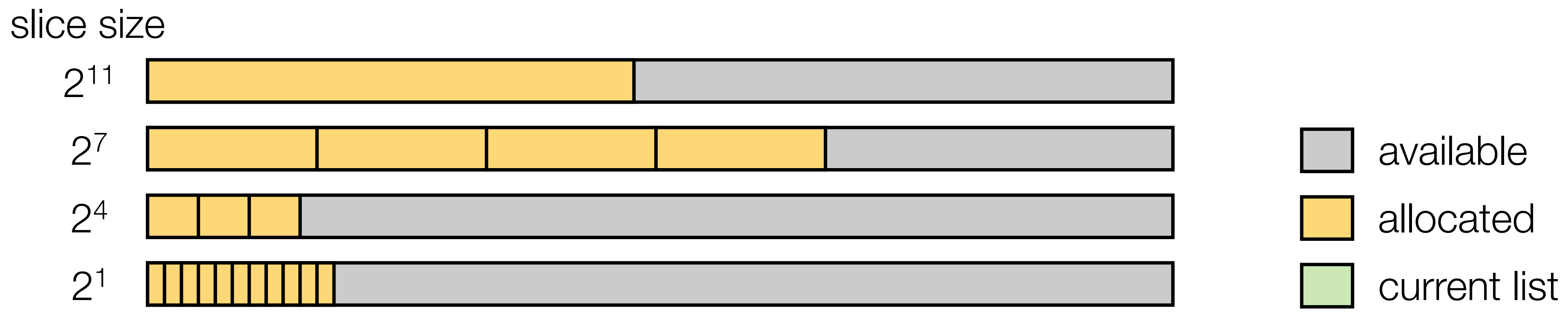
Memory management



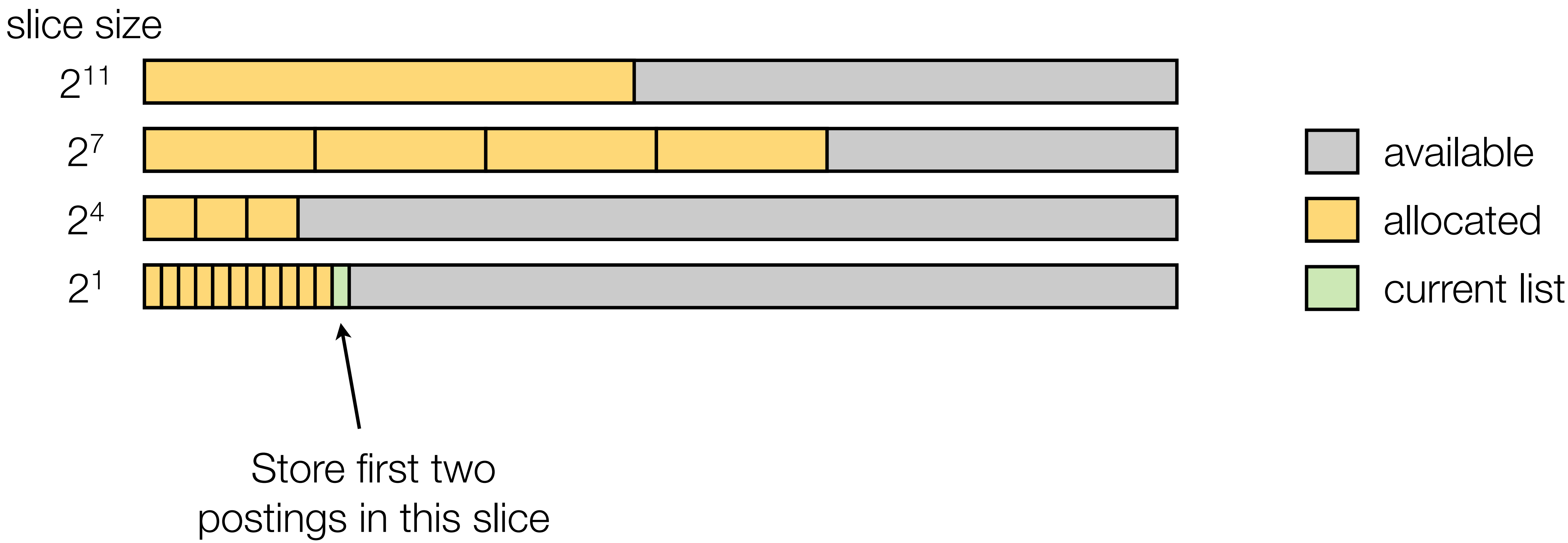
- Slices can be allocated in each pool

- Each pool has a different, but fixed slice size

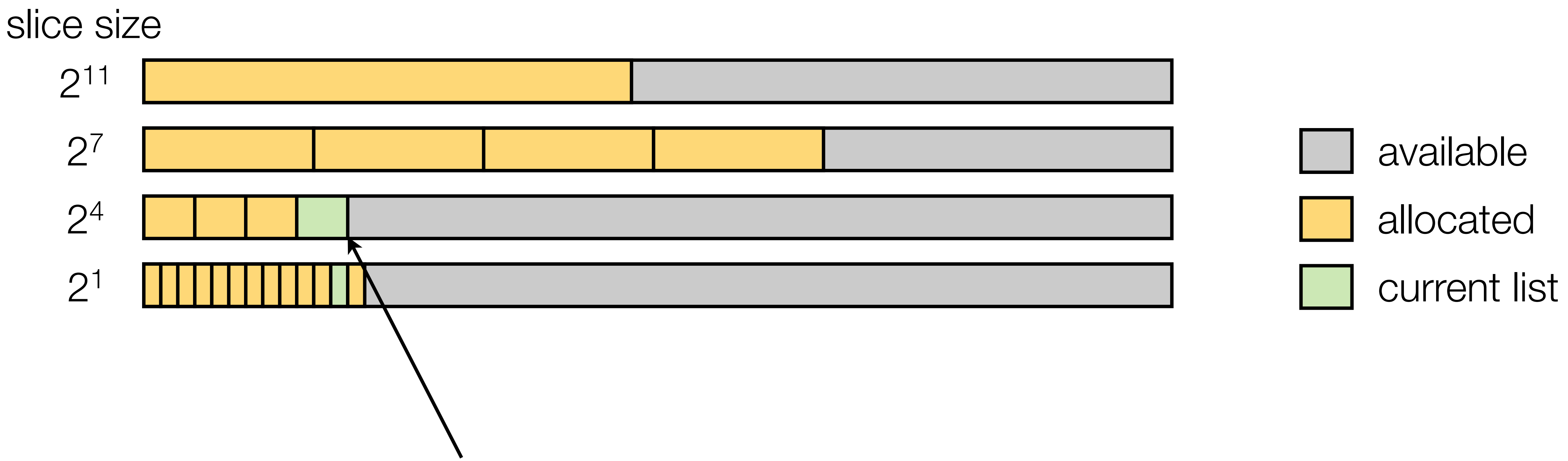
Adding and appending to a list



Adding and appending to a list

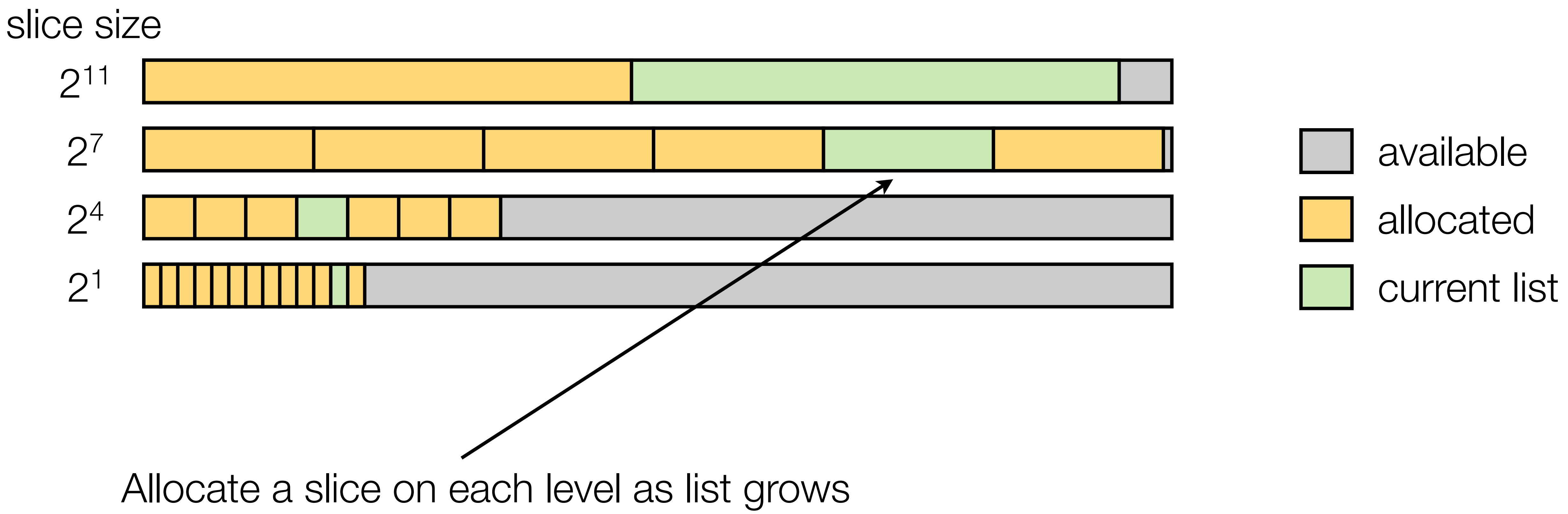


Adding and appending to a list

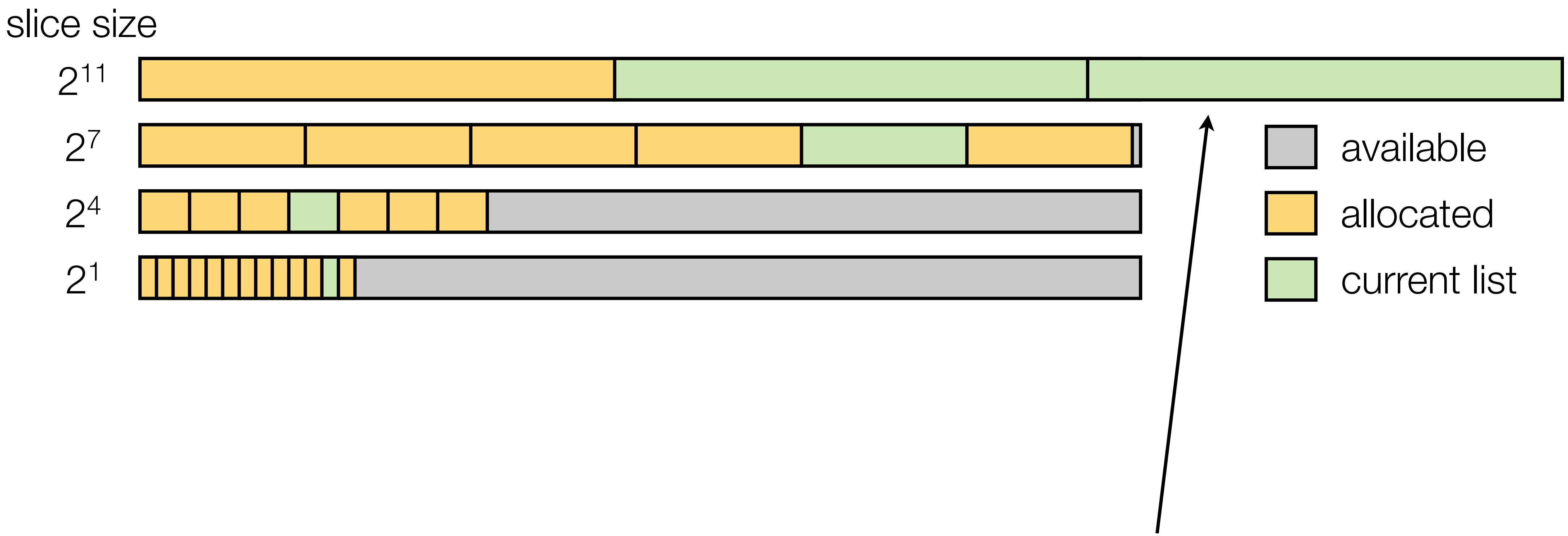


When first slice is full, allocate another one in second pool

Adding and appending to a list

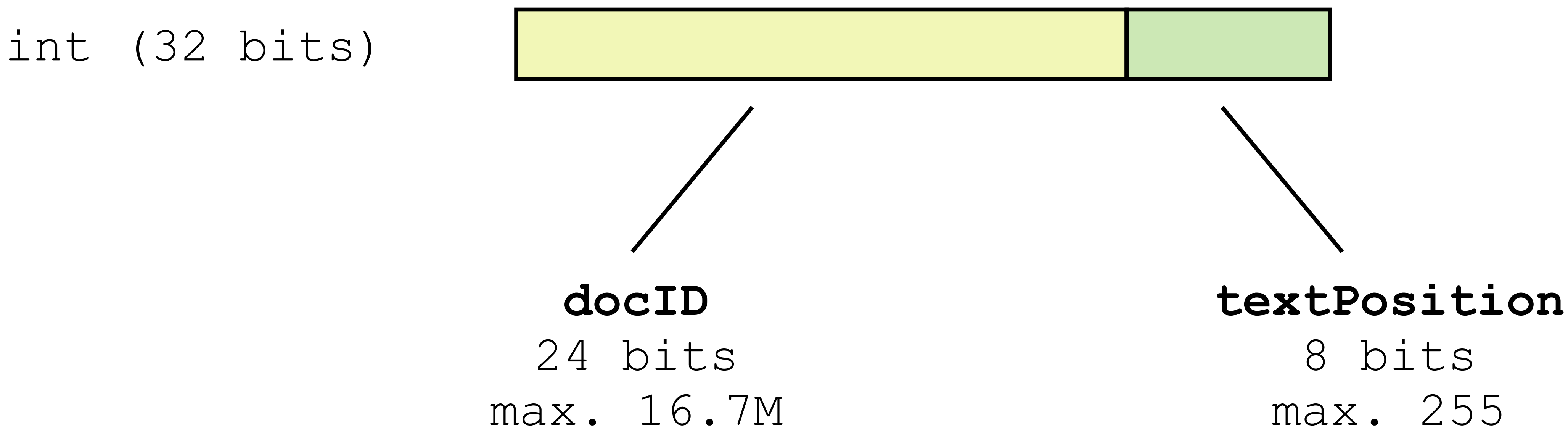


Adding and appending to a list



On upper most level one list can own multiple slices

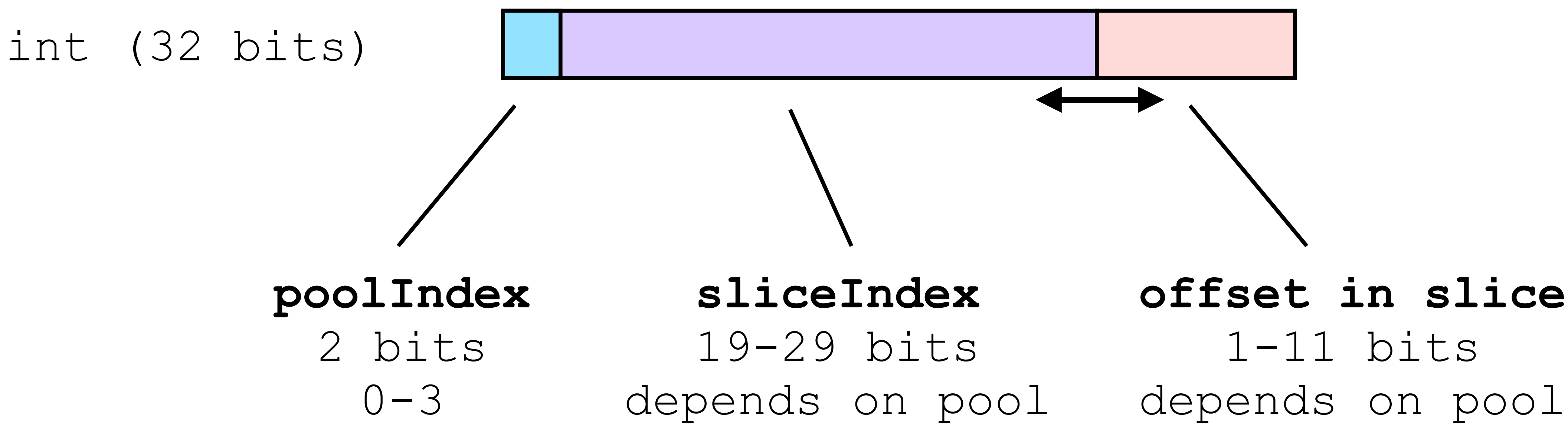
Posting list format



- 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)

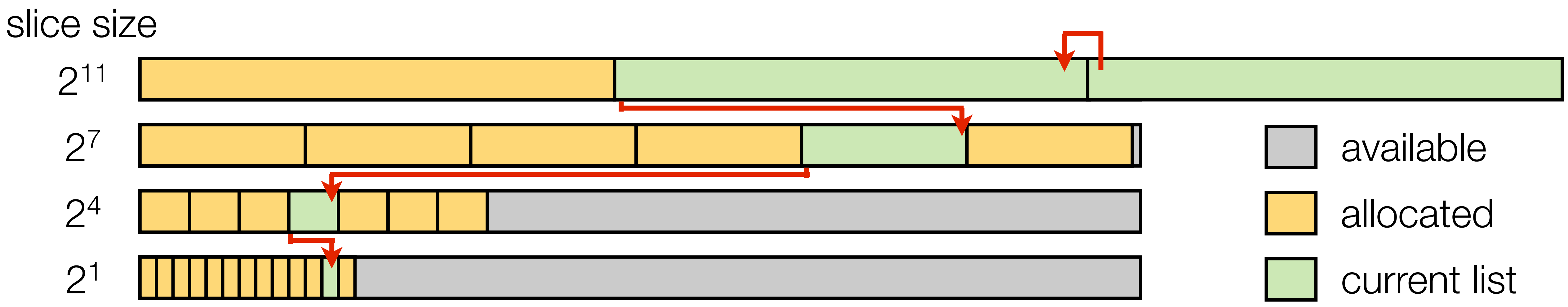
Addressing items

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

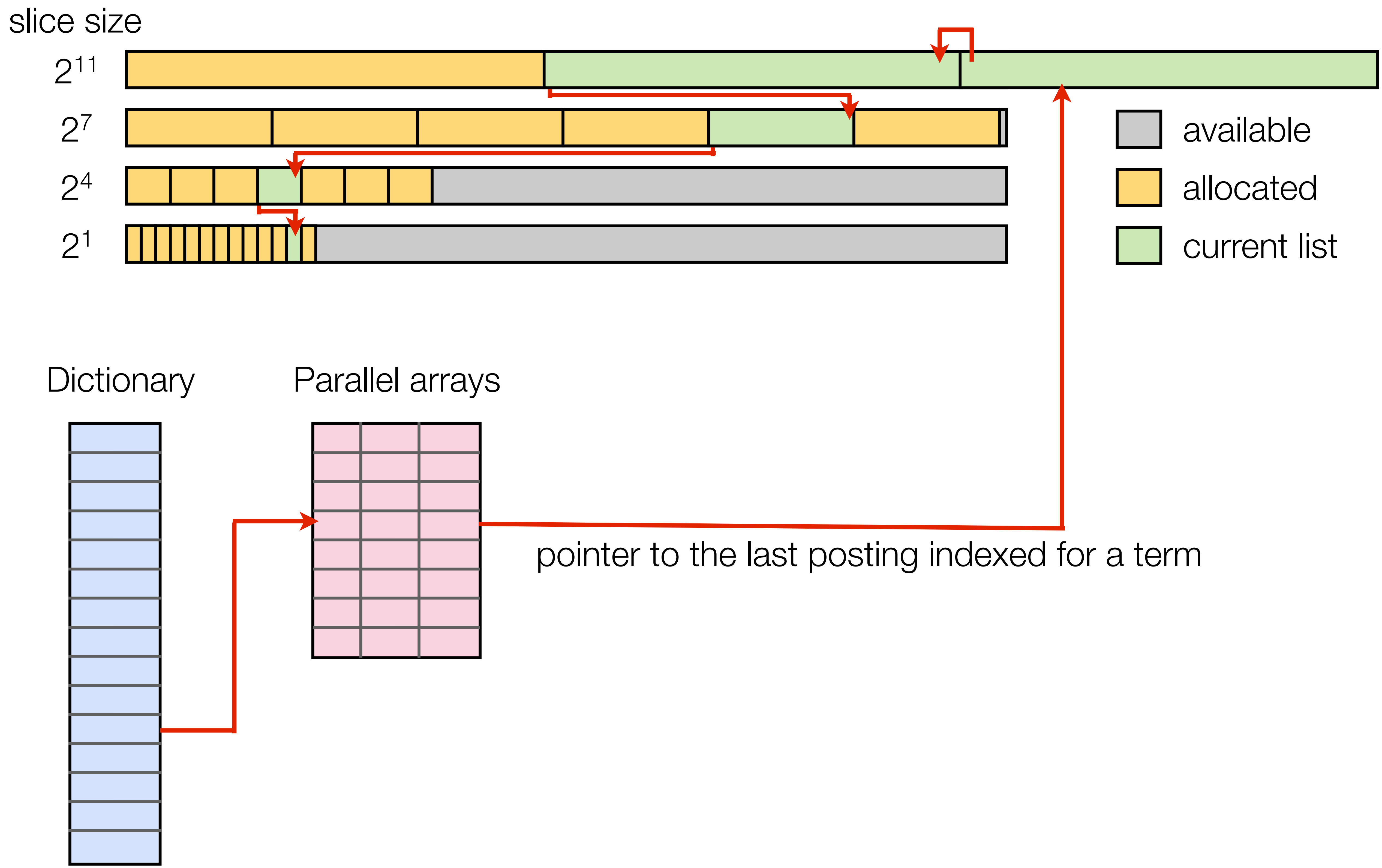


- Nice symmetry: Postings and address pointers both fit into a 32 bit int

Linking the slices



Linking the slices



Concurrency - 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

Concurrency - 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**

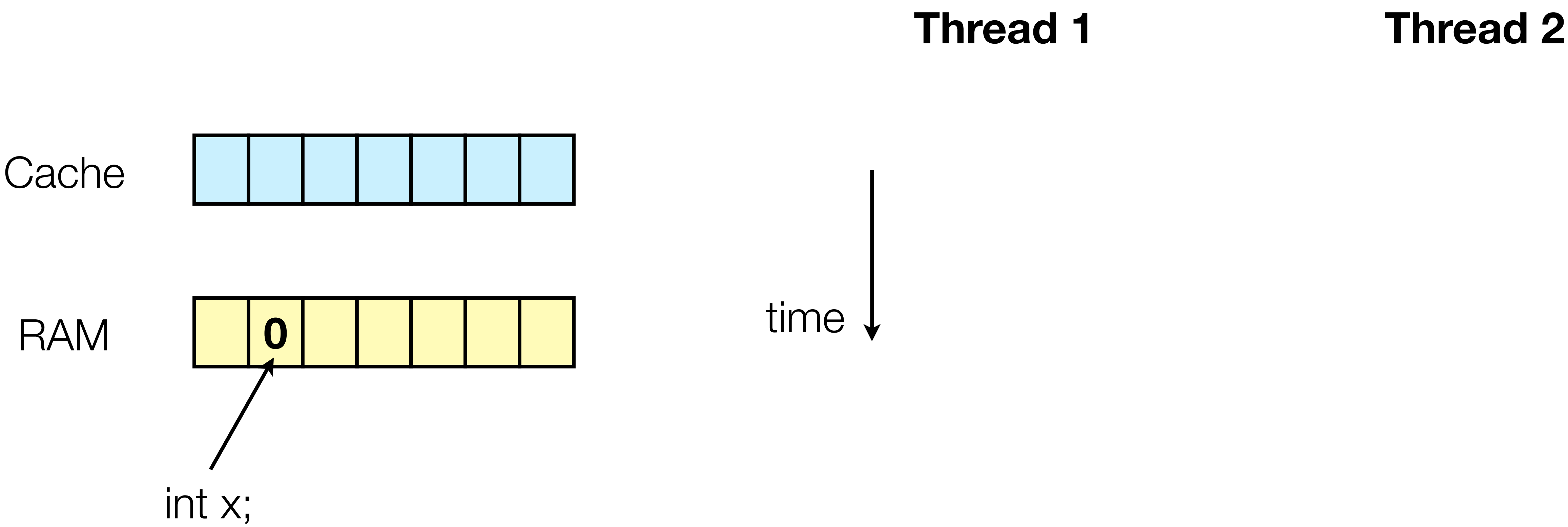
  A write to a volatile field *happens-before* every subsequent read of that same field.

- **Transitivity**

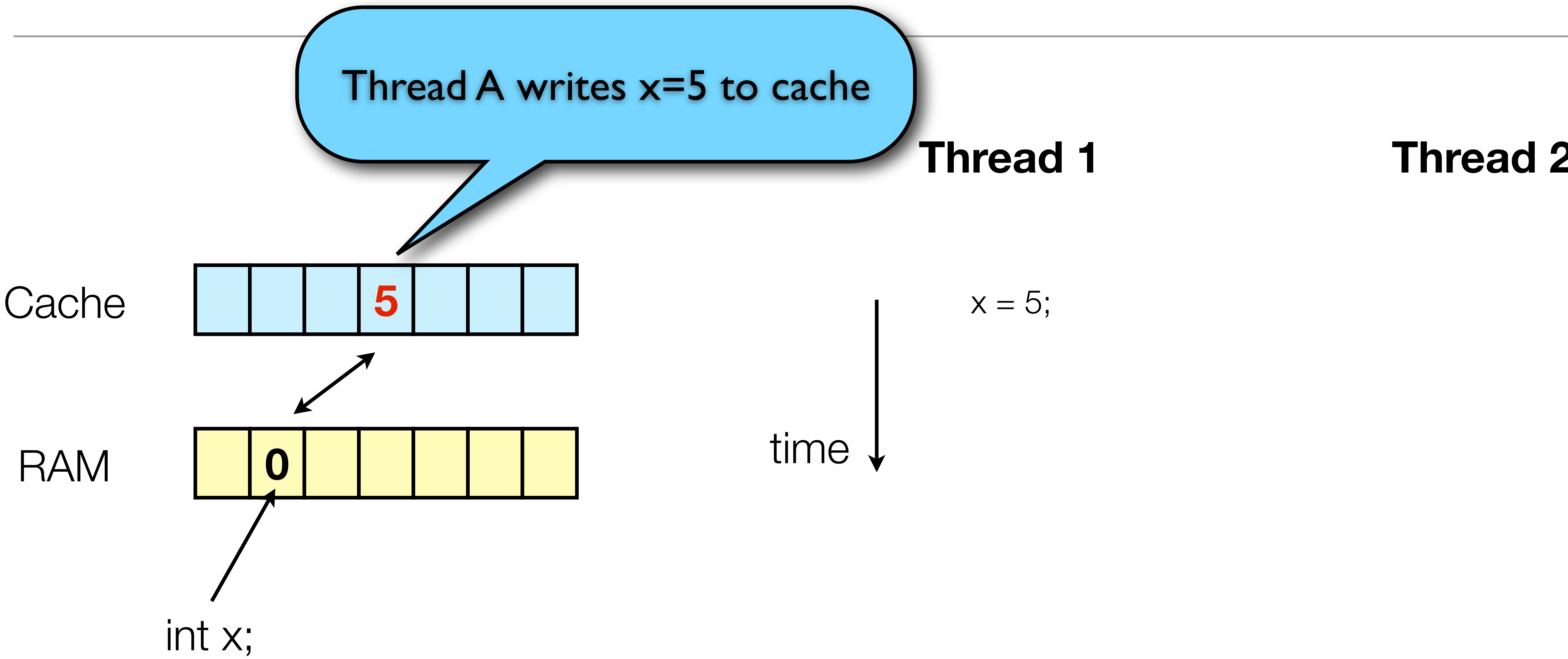
  If A *happens-before* B, and B *happens-before* C, then A *happens-before* C.

* Source: Brian Goetz: Java Concurrency in Practice

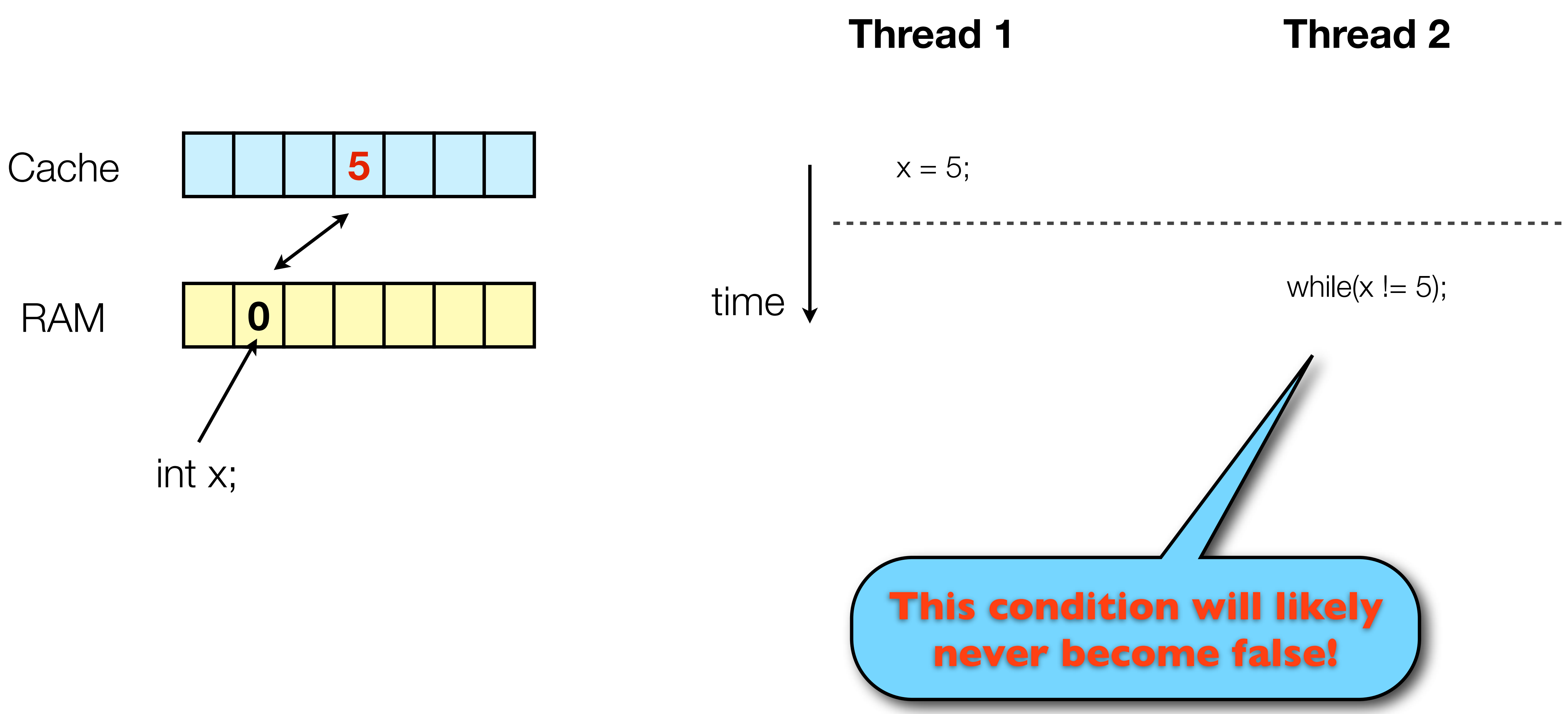
Concurrency



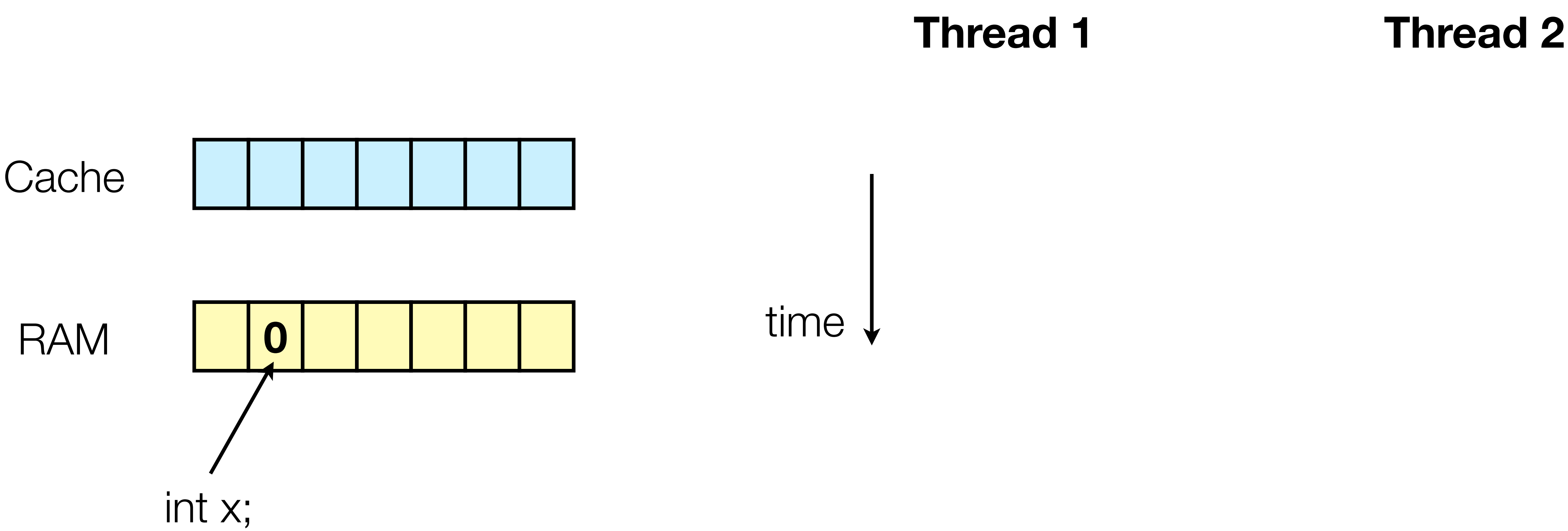
Concurrency



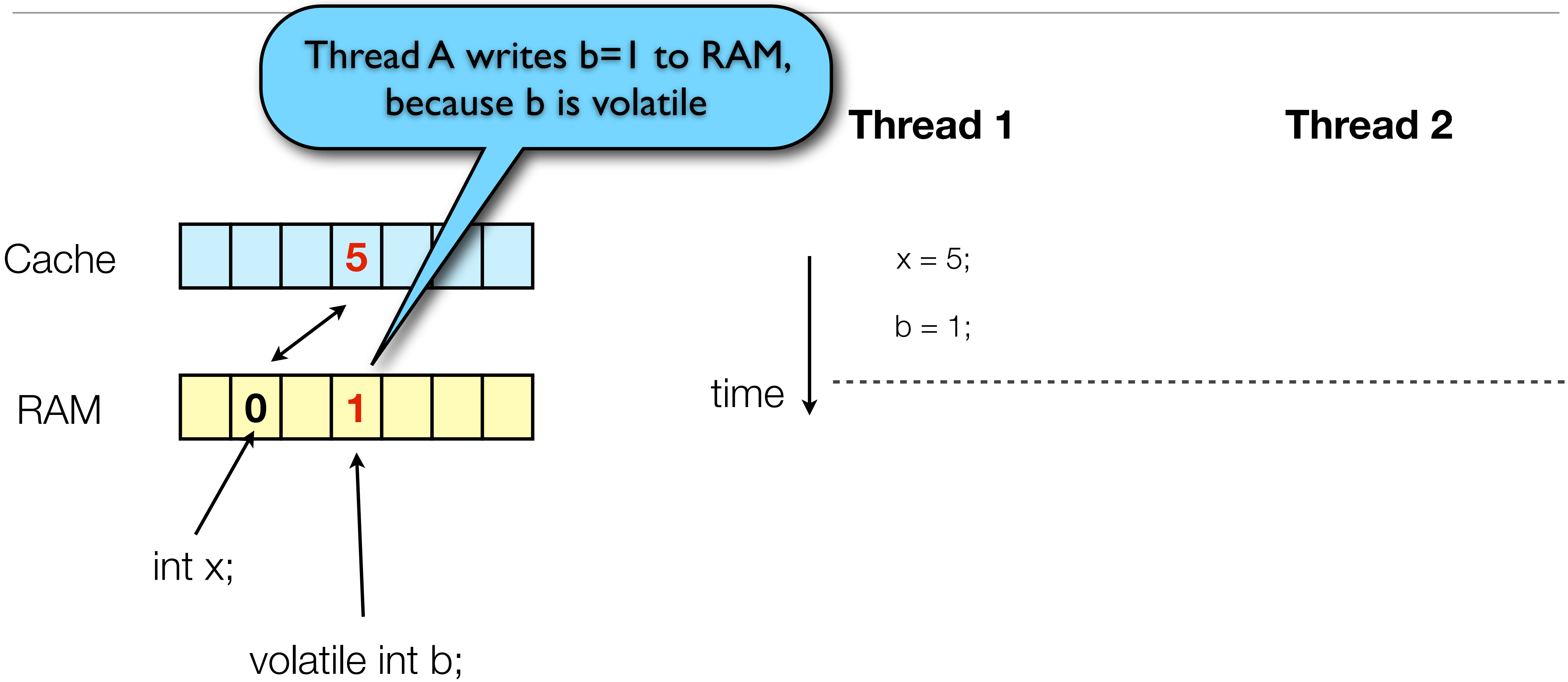
Concurrency



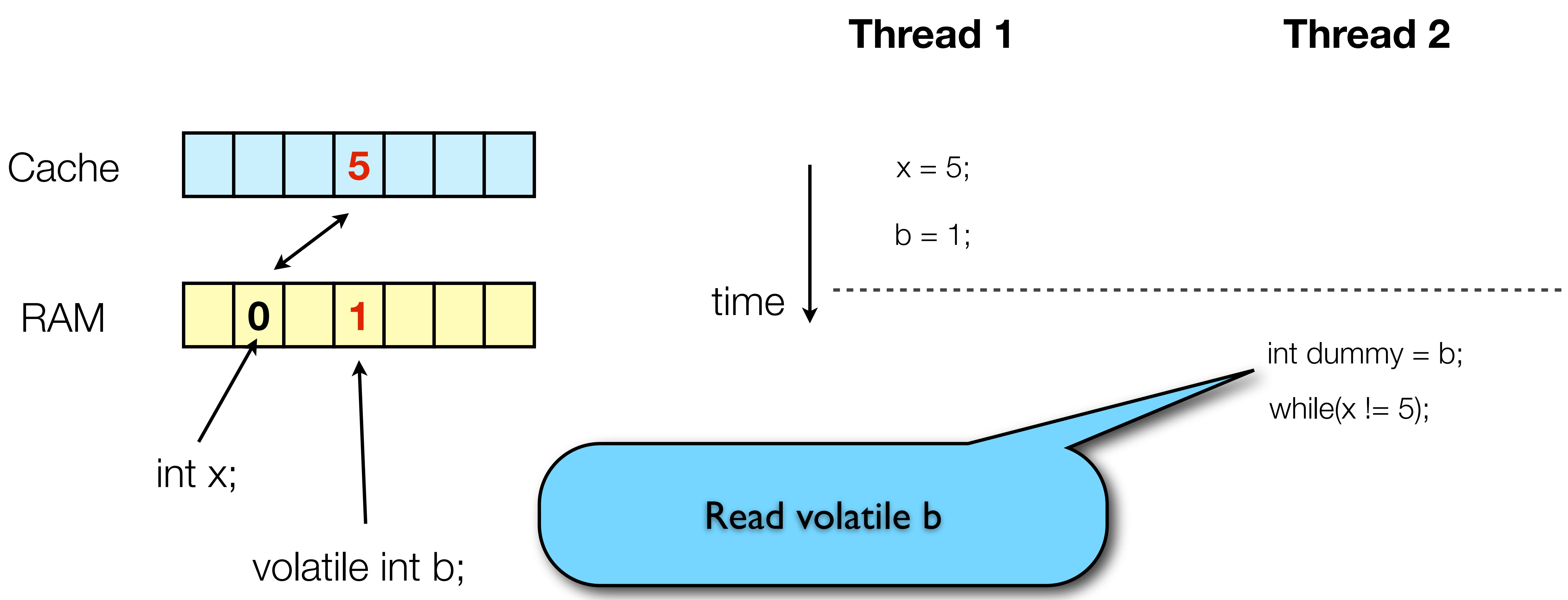
Concurrency



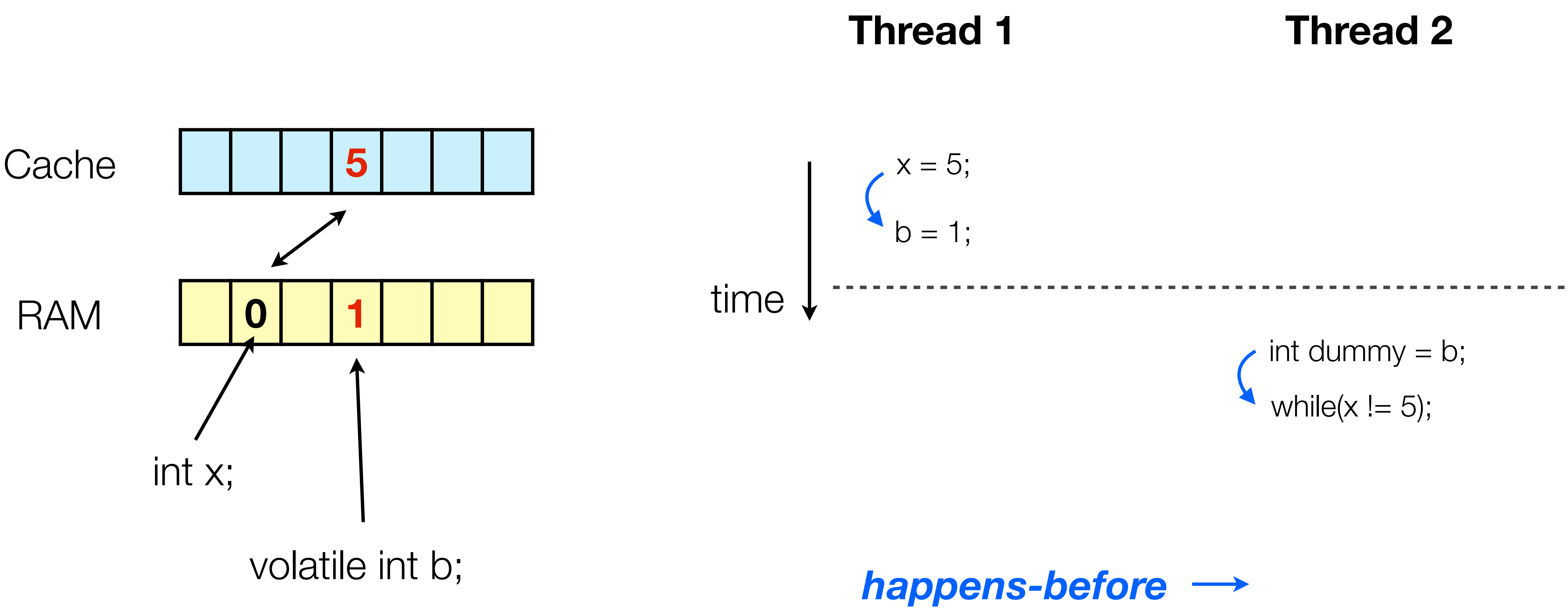
Concurrency



Concurrency

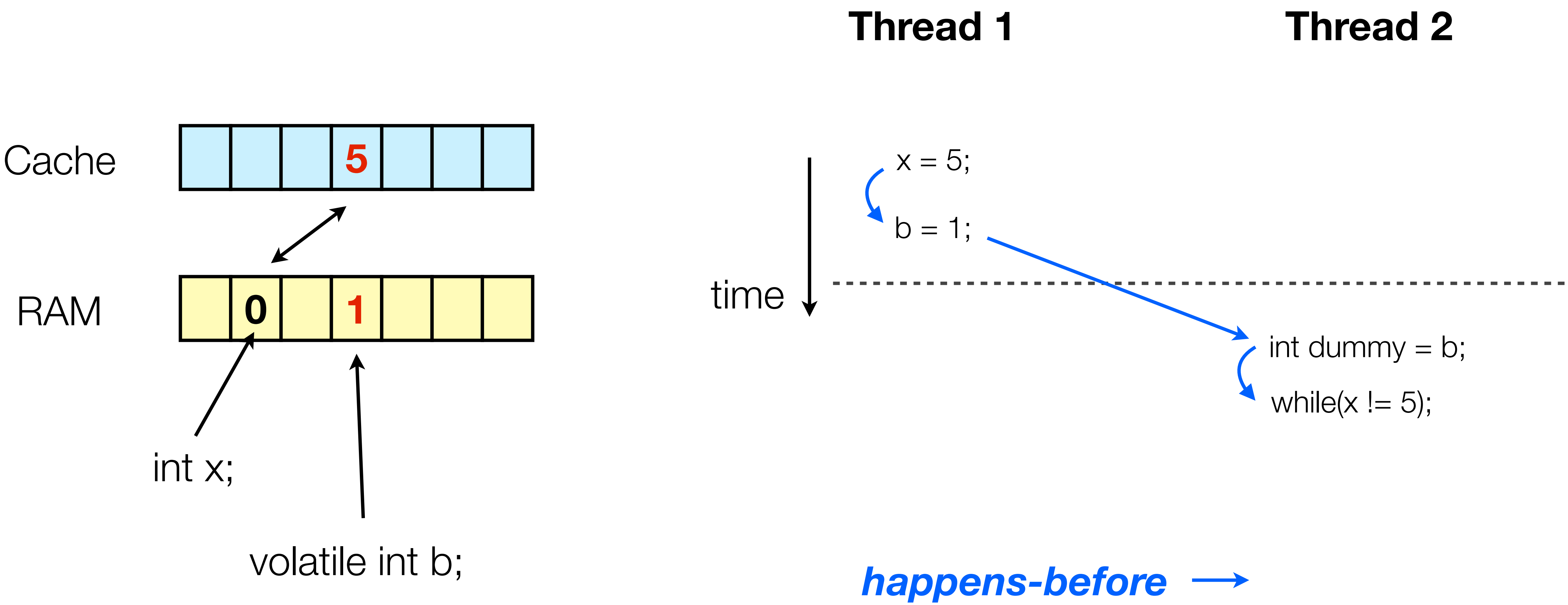


Concurrency



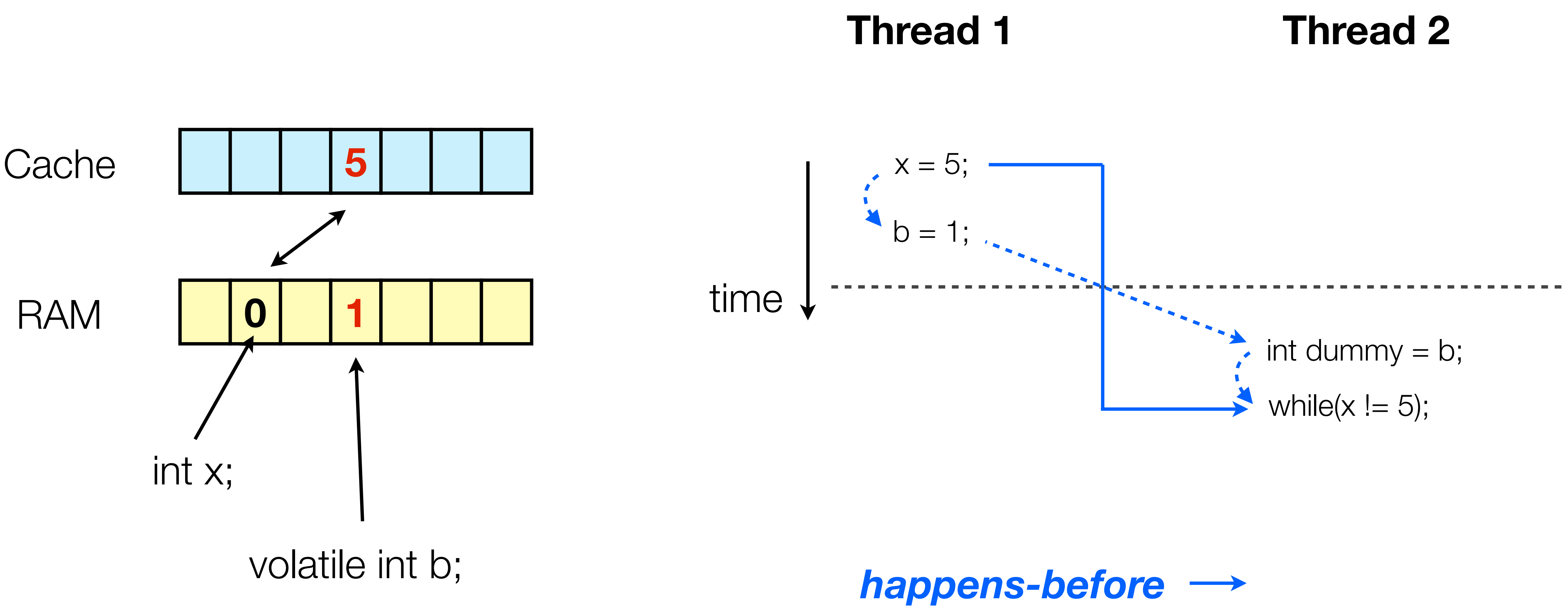
- **Program order rule:** Each action in a thread *happens-before* every action in that thread that comes later in the program order.

Concurrency



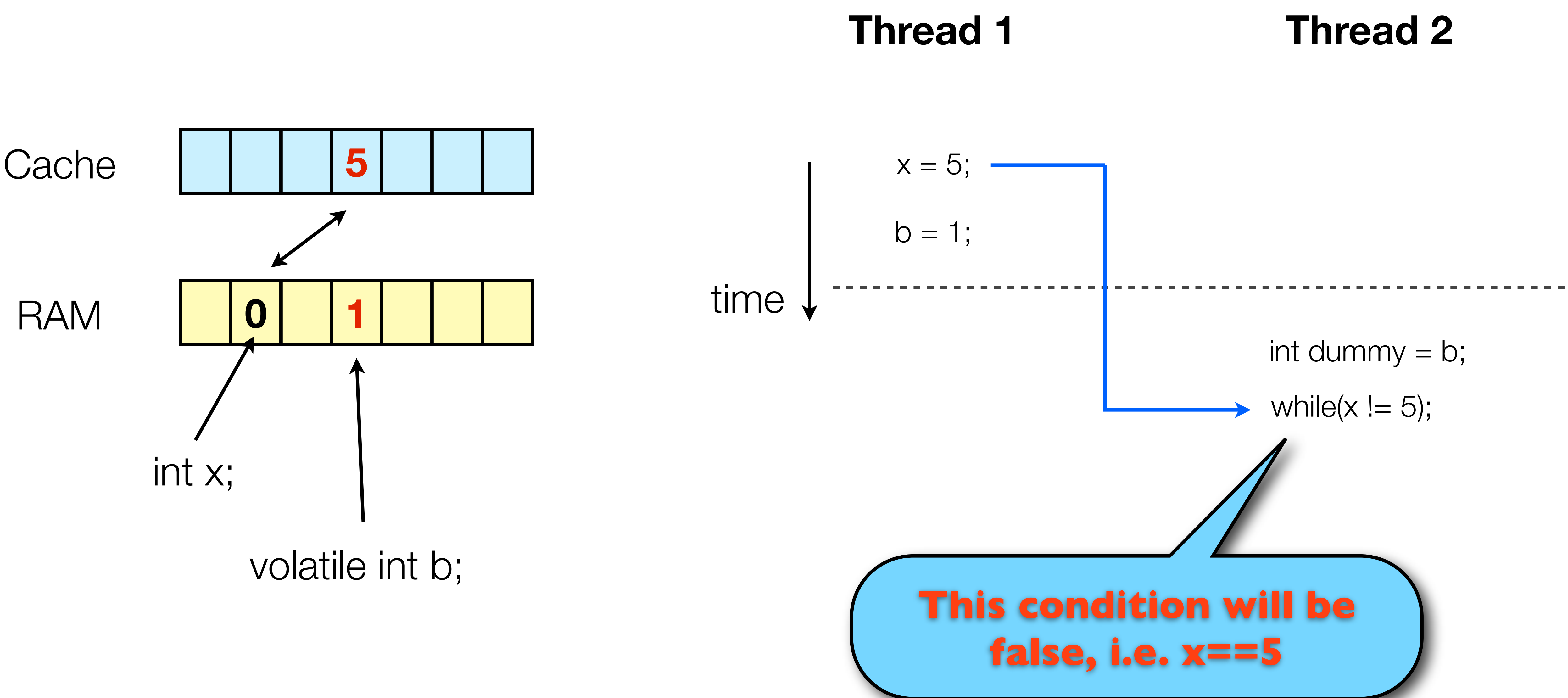
- **Volatile variable rule:** A write to a volatile field *happens-before* every subsequent read of that same field.

Concurrency



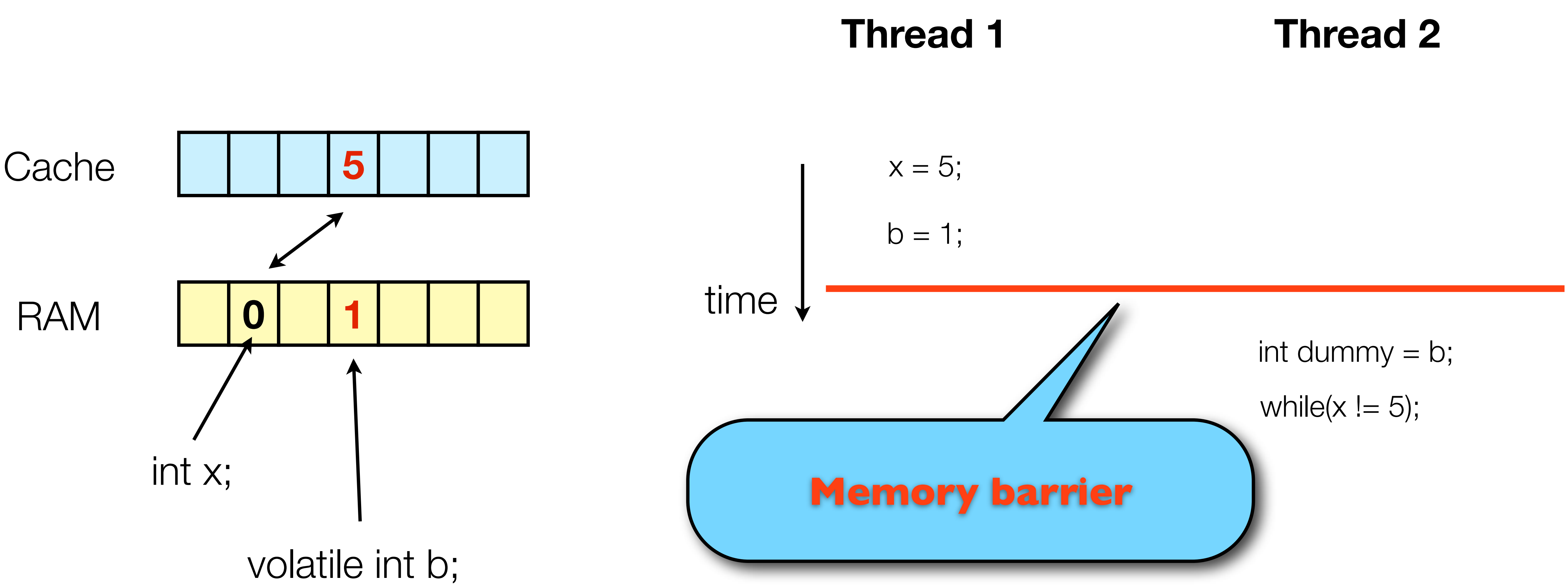
- **Transitivity:** If A *happens-before* B, and B *happens-before* C, then A *happens-before* C.

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.

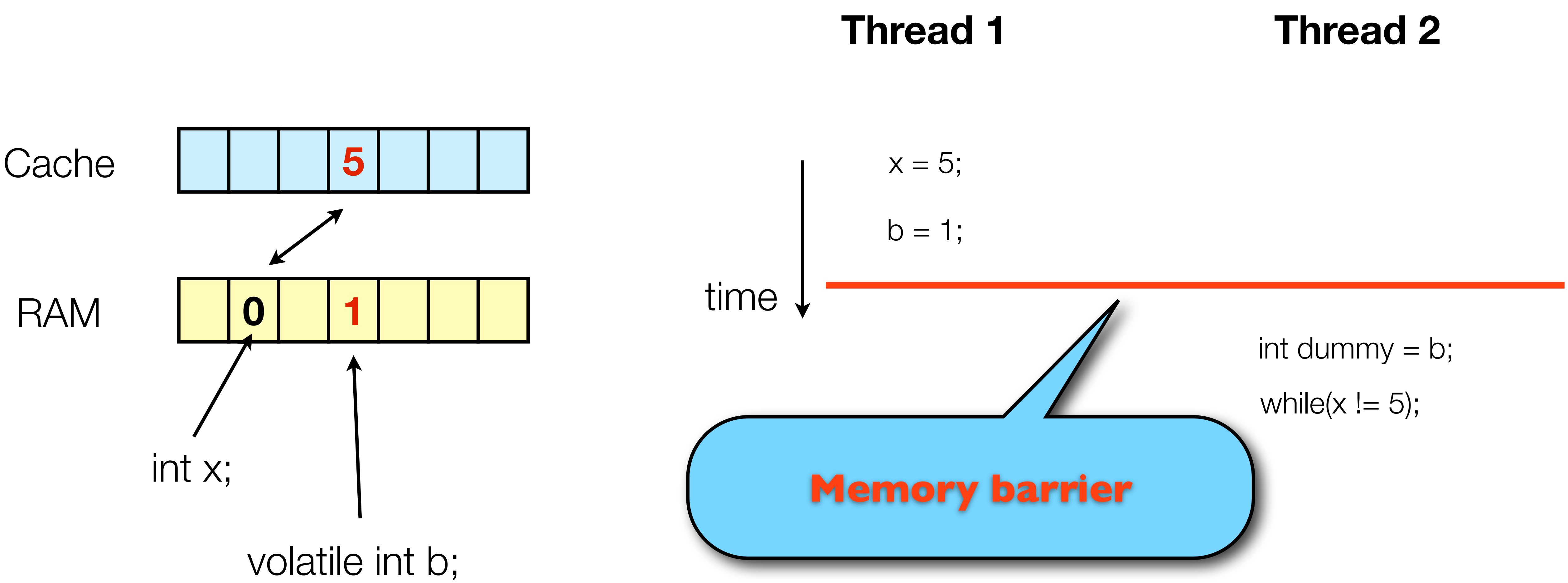
Concurrency



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

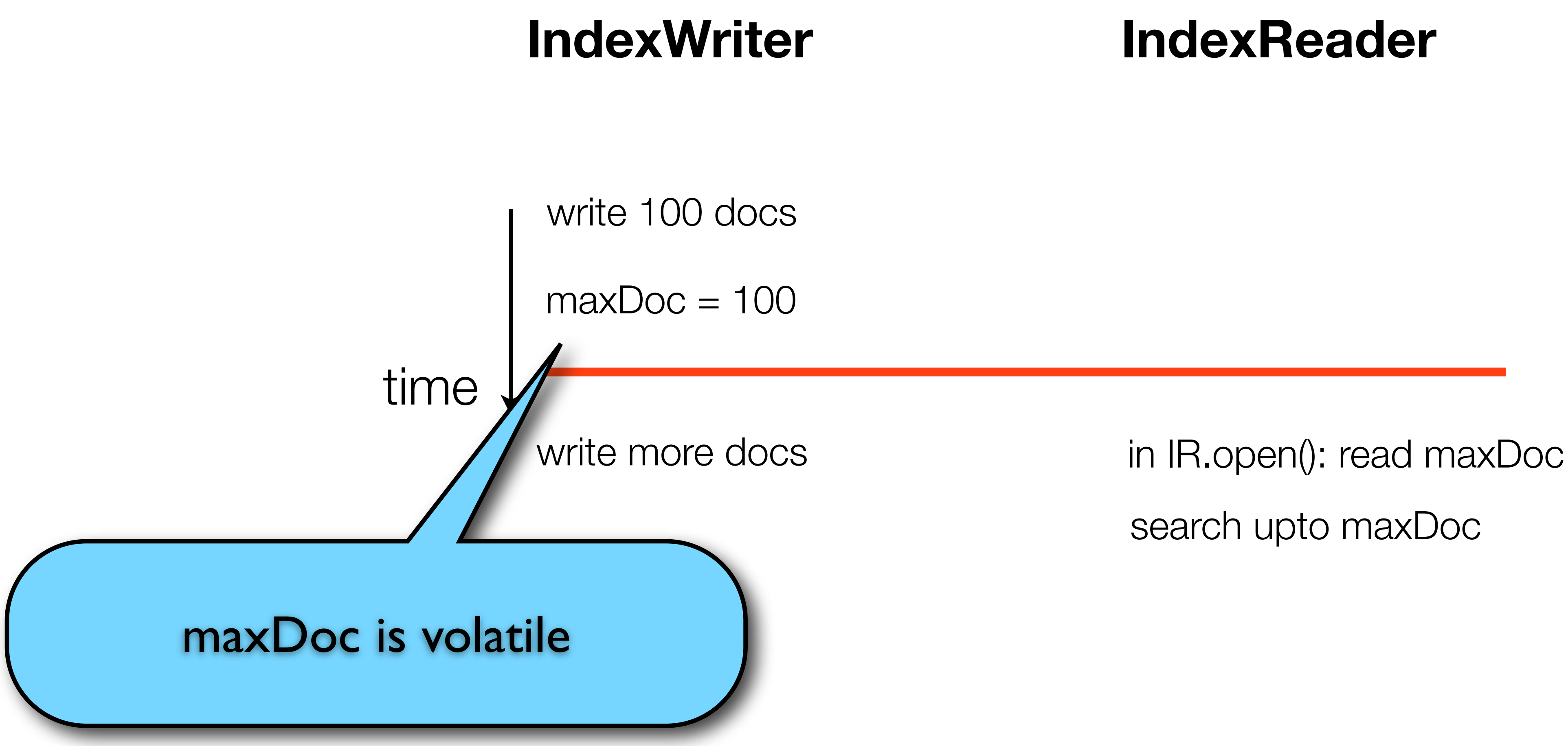
Demo

Concurrency

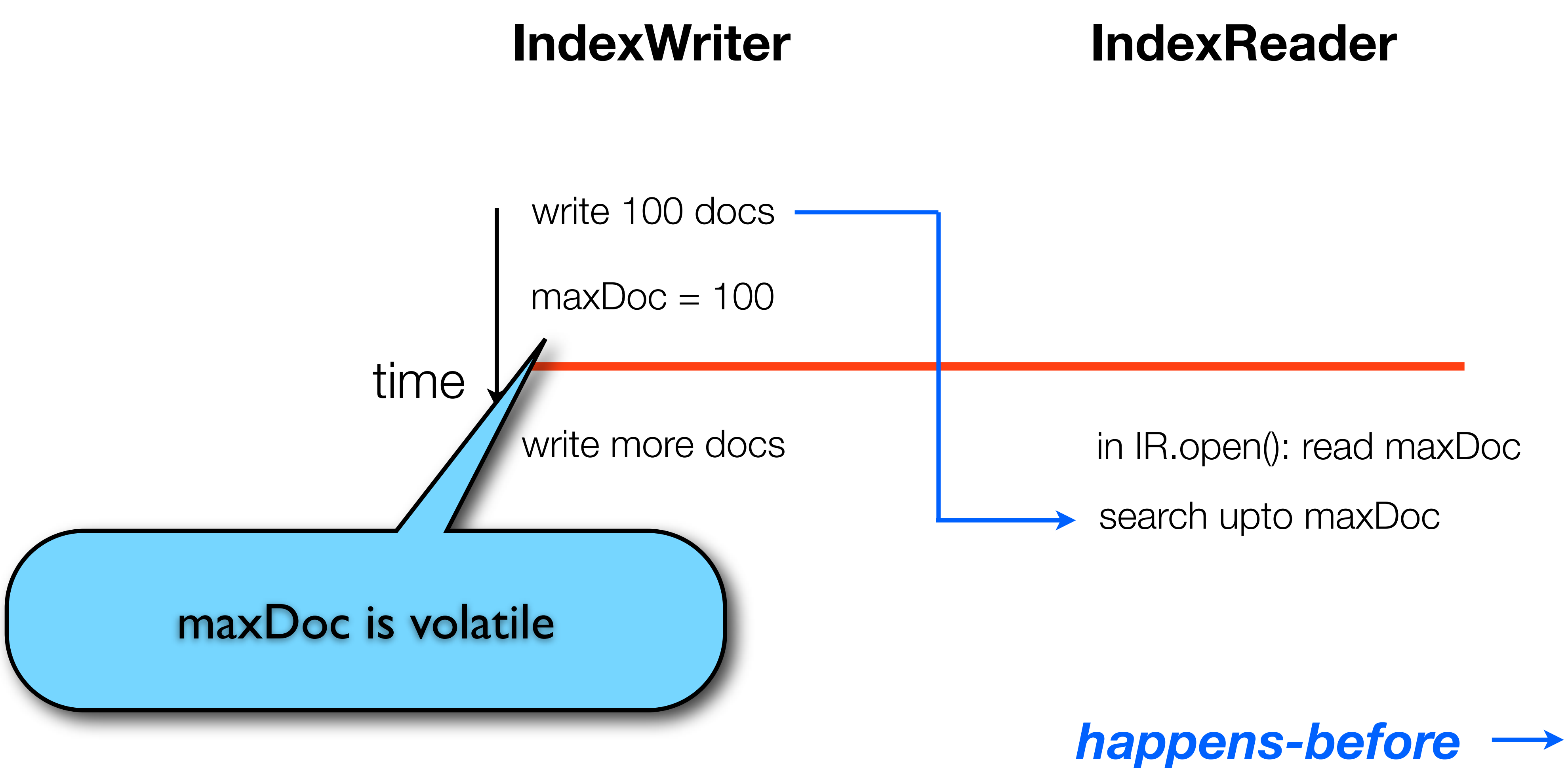


- **Note:** x itself doesn't have to be volatile. There can be many variables like x, but we need only a single volatile field.

Concurrency



Concurrency



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

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

Realtime Search @twitter

Agenda

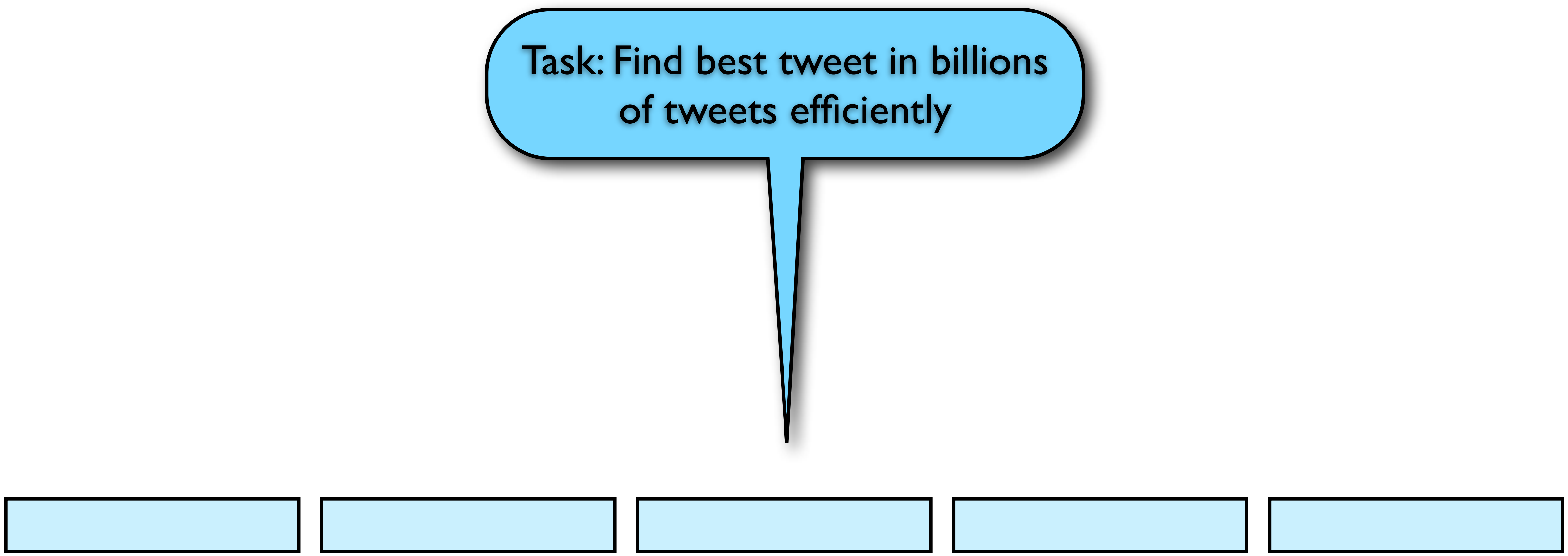
- Introduction
- Search Architecture
- Inverted Index 101
- Memory Model & Concurrency
- Top Tweets

Top Tweets

Signals

- Query-dependent
  - E.g. Lucene text score, language
- Query-independent
  - Static signals (e.g. text quality)
  - Dynamic signals (e.g. retweets)
  - Timeliness

Signals



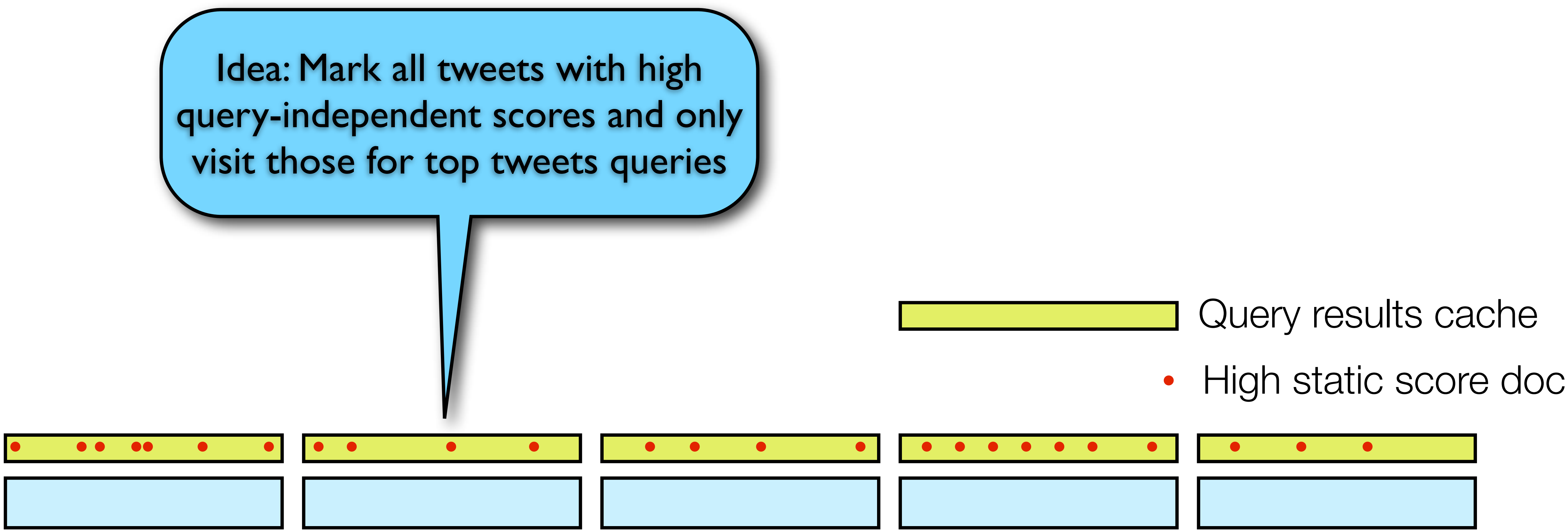
Many Earlybird segments (8M documents each)

Signals

- For top tweets we can’t early terminate as efficiently
- Scoring and ranking billions of tweets is impractical

Many Earlybird segments (8M documents each)

Query cache



Many Earlybird segments (8M documents each)

Query cache

- A background thread periodically wakes up, executes queries, and stores the results in the per-segment cache

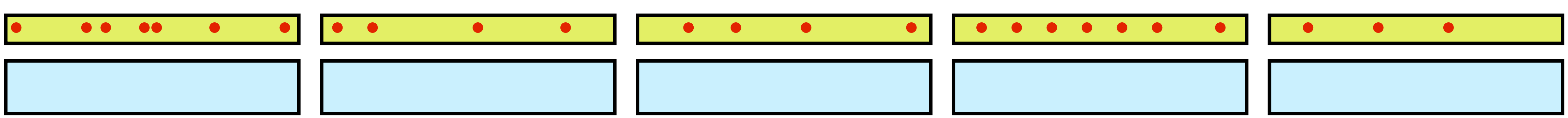
- Rewriting queries

    - User query: q = ‘lucene’

    - Rewritten: q’ = ‘lucene AND cached_filter:toptweets’

This clause will be executed as a Lucene ConstantScoreQuery that wraps a BitSet or SortedVIntList

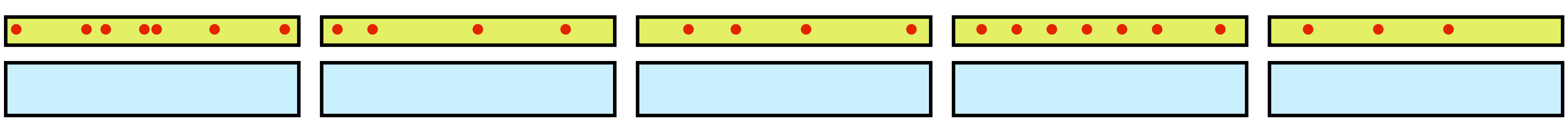
- Efficient skipping over tweets with low query-independent scores



Many Earlybird segments (8M documents each)

Query cache

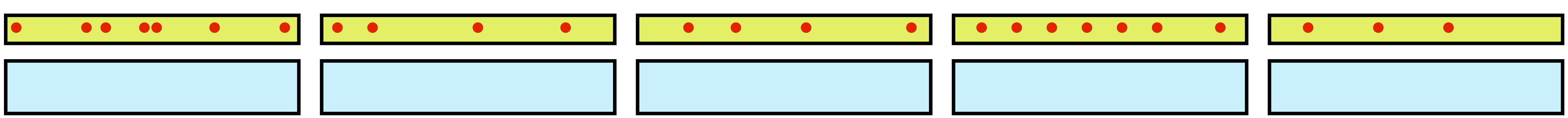
- A background thread periodically wakes up, executes queries, and stores the results in the per-segment cache
- Configurable per cached query:
  - Result set type: BitSet, SortedVIntList
  - Execution schedule
  - Filter mode: cache-only or hybrid



Many Earlybird segments (8M documents each)

Query cache

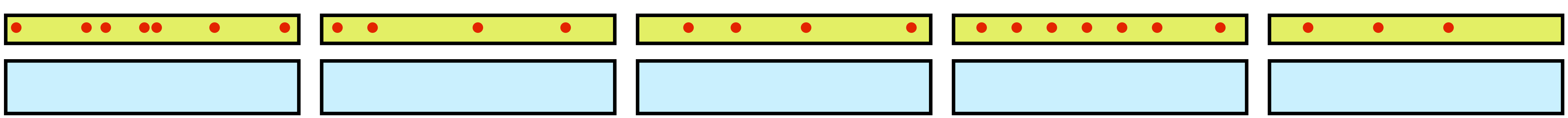
- **Result set type: BitSet, SortedVIntList**
  - BitSet for results with many hits
  - SortedVIntList for very sparse results



Many Earlybird segments (8M documents each)

Query cache

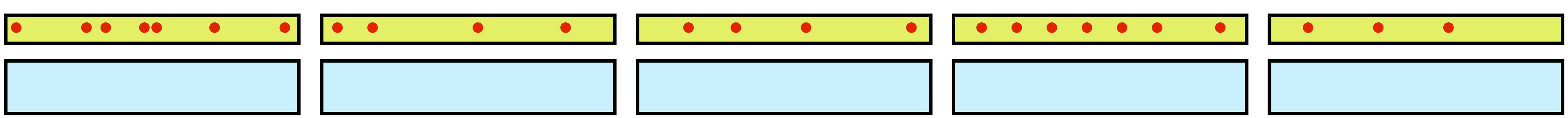
- **Execution schedule**
    - Per segment: Sleep time between refreshing the cached results



Many Earlybird segments (8M documents each)

Query cache

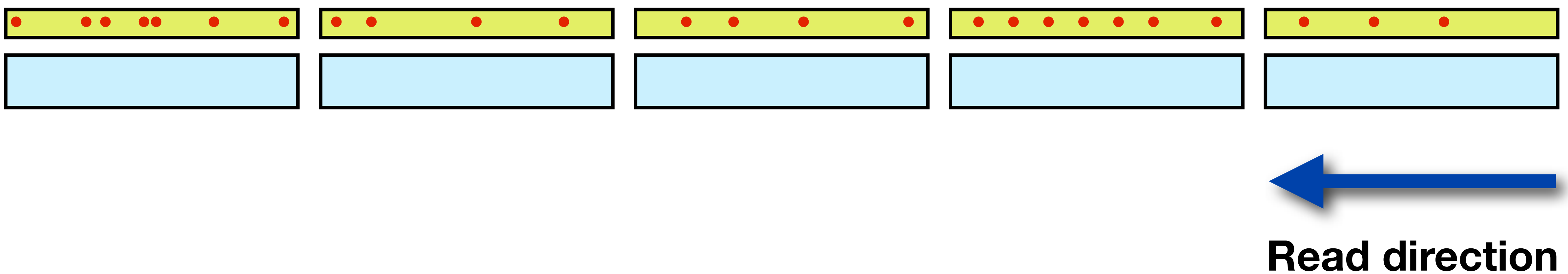
- **Filter mode: cache-only or hybrid**



Many Earlybird segments (8M documents each)

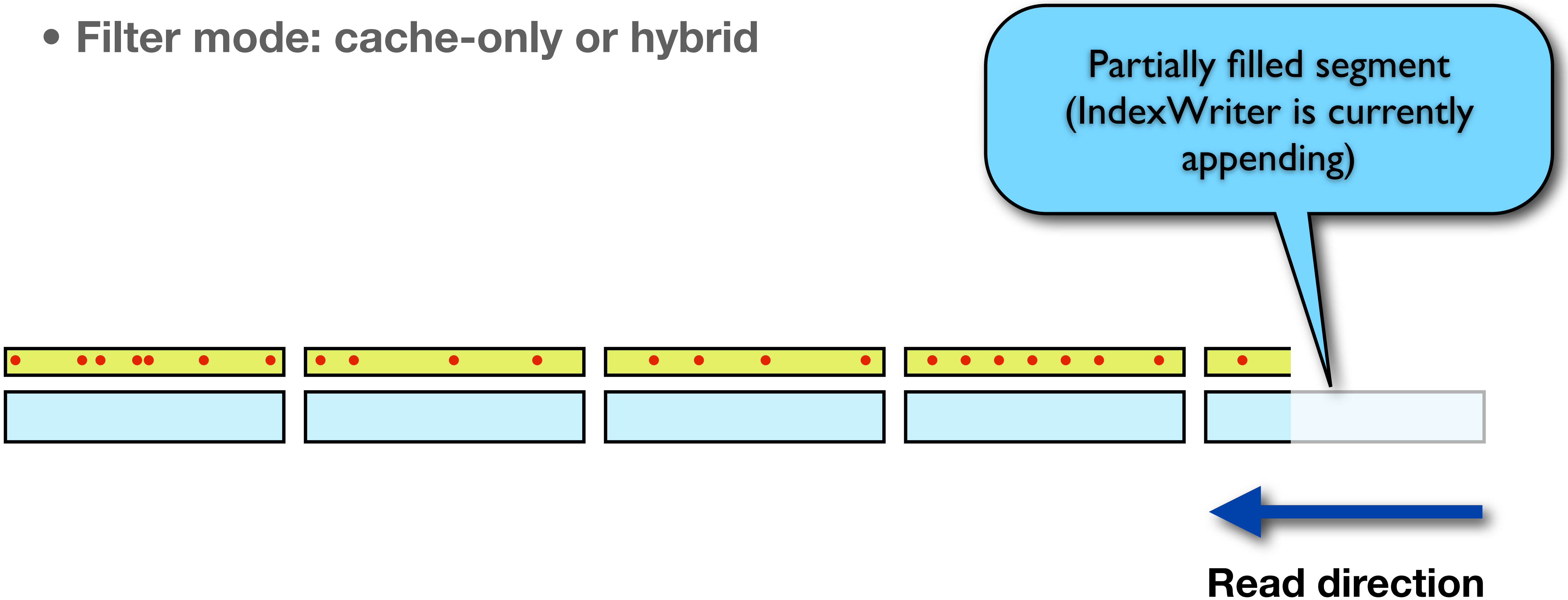
Query cache

- **Filter mode: cache-only or hybrid**



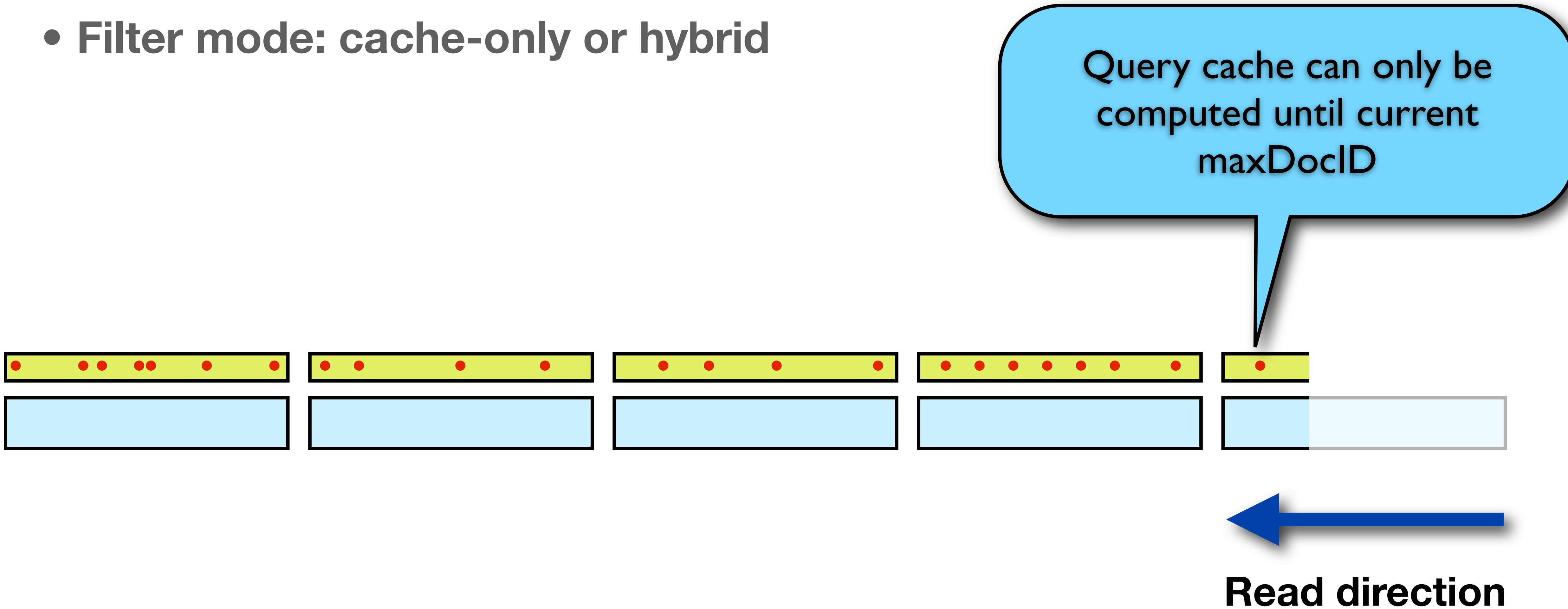
Query cache

- **Filter mode: cache-only or hybrid**



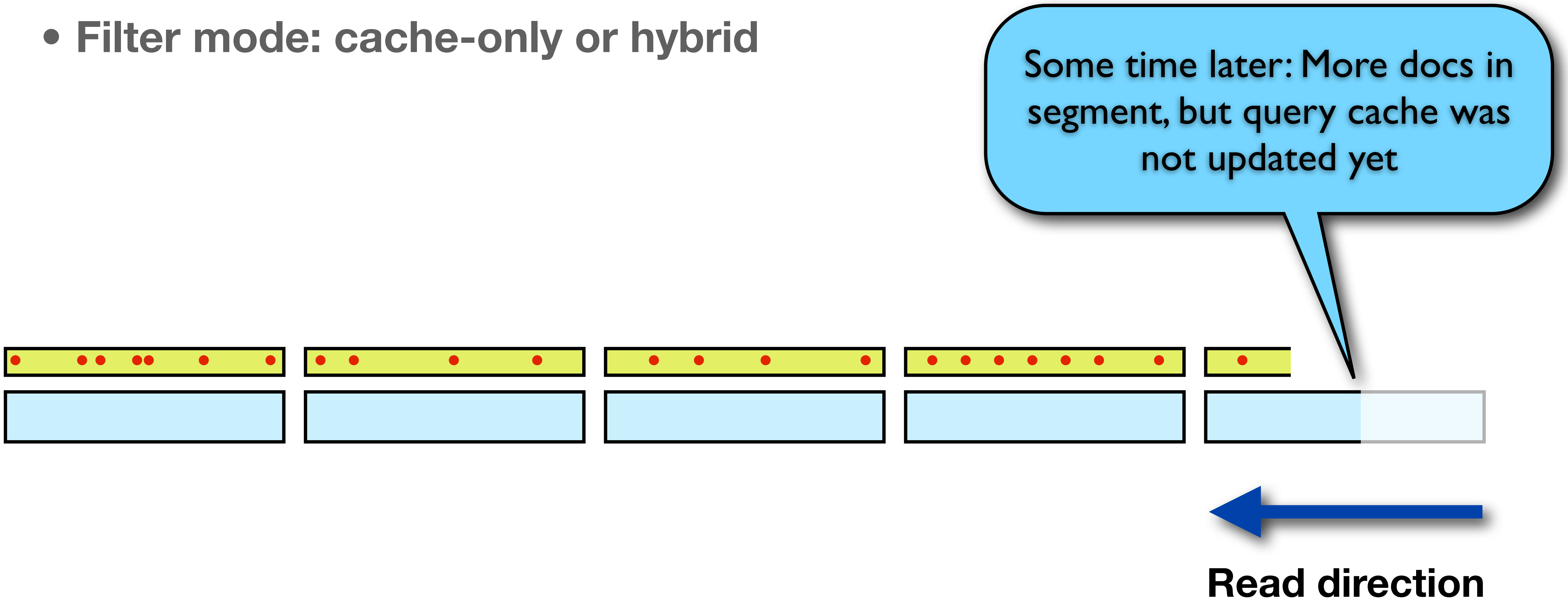
Query cache

- **Filter mode: cache-only or hybrid**



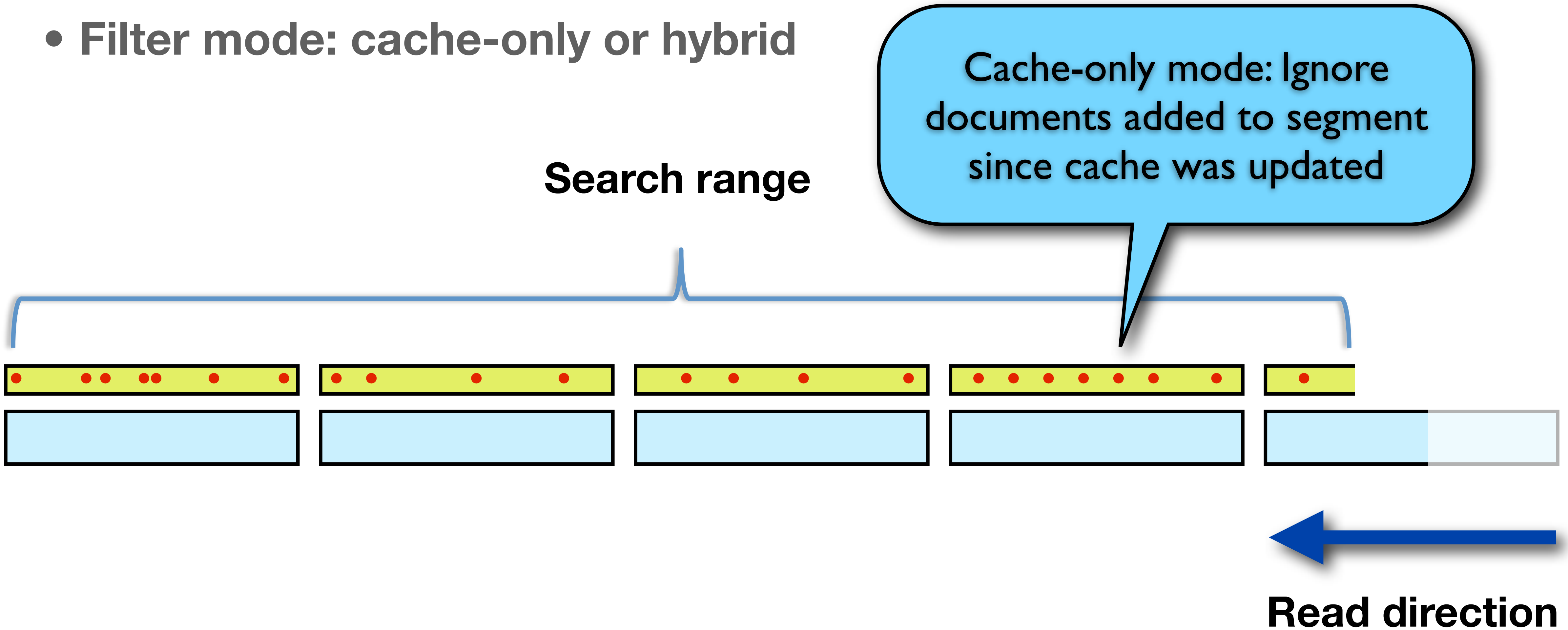
Query cache

- **Filter mode: cache-only or hybrid**



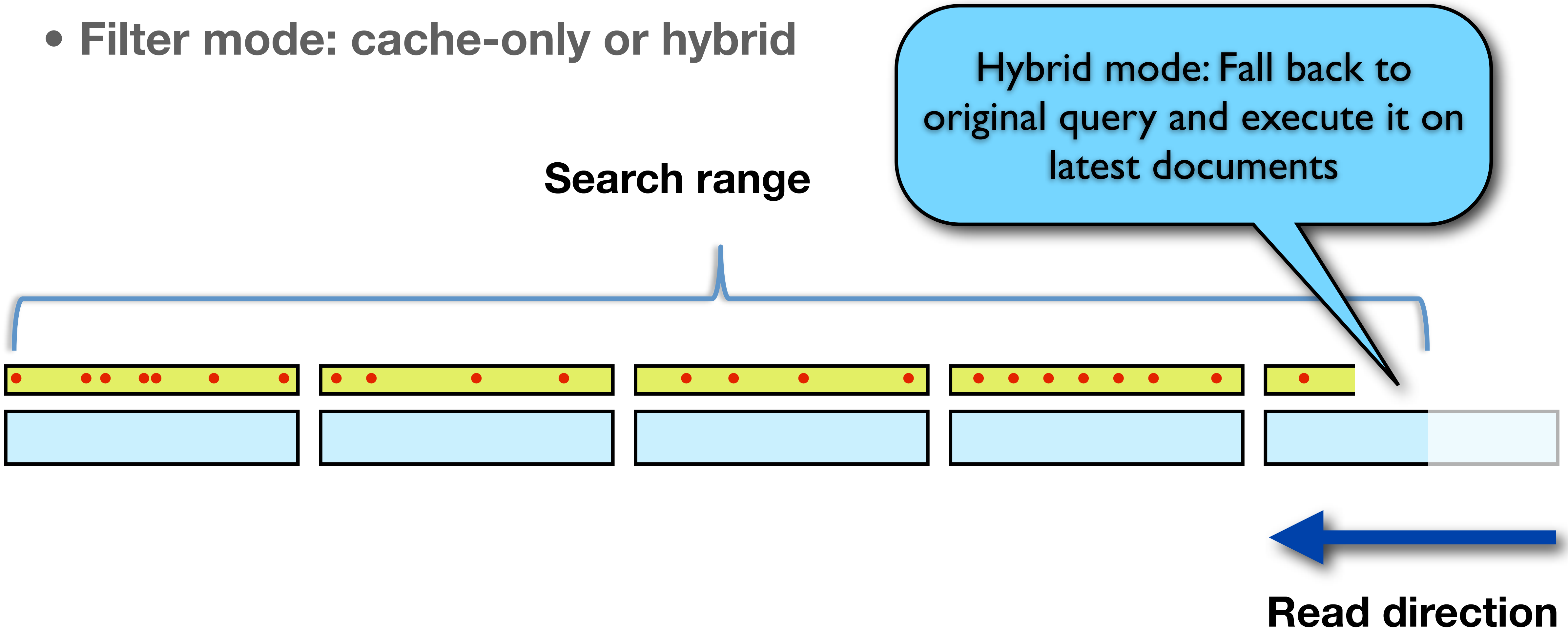
Query cache

- **Filter mode: cache-only or hybrid**



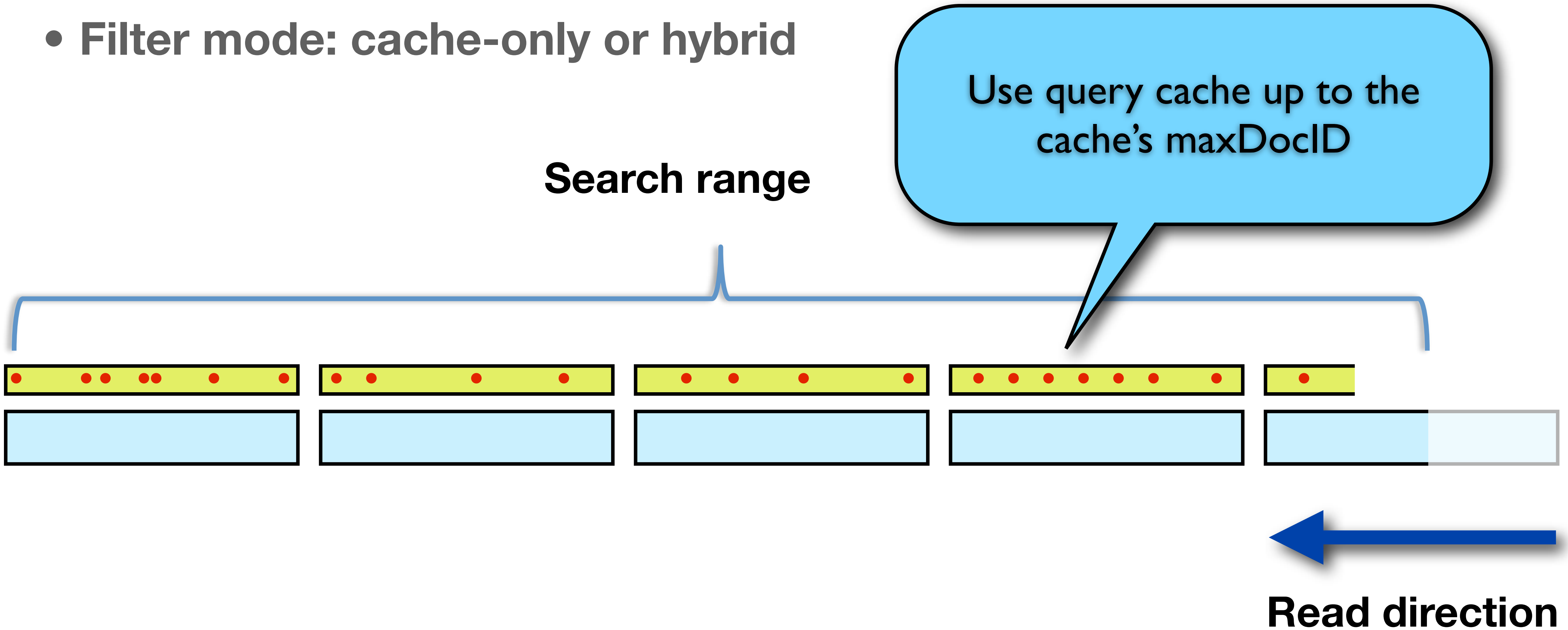
Query cache

- Filter mode: cache-only or hybrid



Query cache

- Filter mode: cache-only or hybrid



Query result cache

```
filterName: toptweets
query: '(* [score_filter toptweets 0.21 1.0])'
resultType: BitSet
cacheModeOnly: true
schedule:
- {segment: 0, seconds: 300}
- {segment: 1, seconds: 600}
- {segment: 2, seconds: 3600}
# only twice per day for older segments
- {segment: 4, seconds: 43200}
```

query cache config yml file

Many Earlybird segments (8M documents each)

Questions?