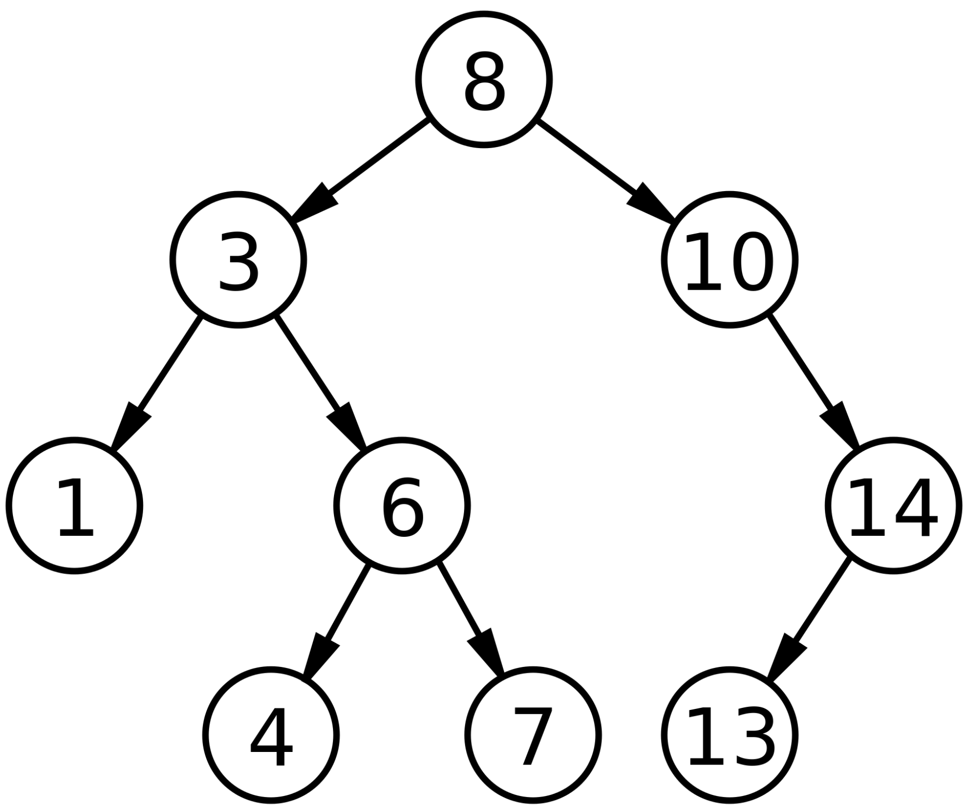
[544] Cassandra Storage Engine

Tyler Caraza-Harter

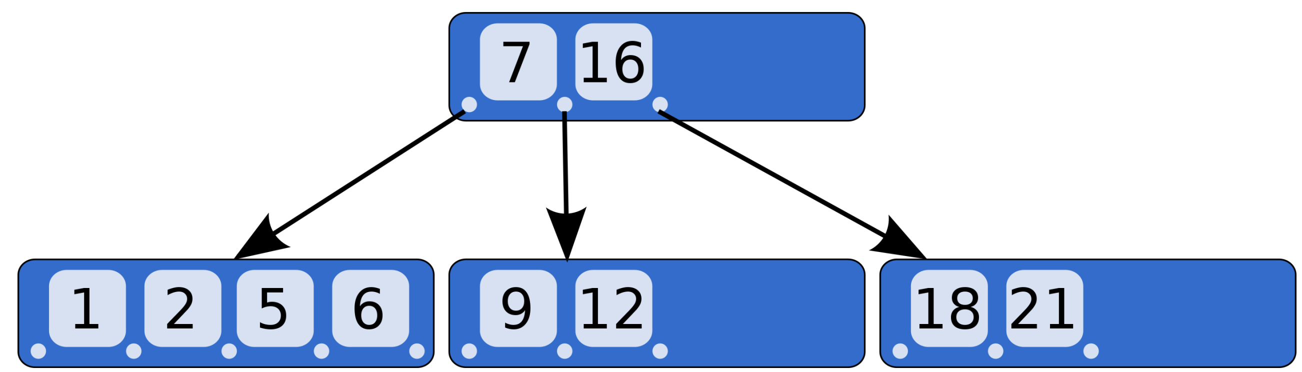
Binary Search Trees, B-trees

Binary Search Tree



https://en.wikipedia.org/wiki/Binary_search_tree

B-Tree



https://en.wikipedia.org/wiki/B-tree

MySQL example:
CREATE INDEX loan_amount_idx
USING BTREE ON loans(loan_amount)

B-Trees:

- most popular DB index data structure
- fast reads, slower for writes
- more in CS 564

Log Structured Merge Trees (LSMs)

Performance:

- faster writes (ALWAYS sequential)
- writes create background work to complete later
- slower reads (for single value, at least)

Single-node DBs and K/V stores:

- LevelDB
- RocksDB
- SQLite4

Distributed DBs:

- BigTable
- HBase
- Cassandra

Outline: Cassandra Storage Engine

Writing Data: Buffering and Logging

- in general
- Cassandra

Storing Data: SSTables

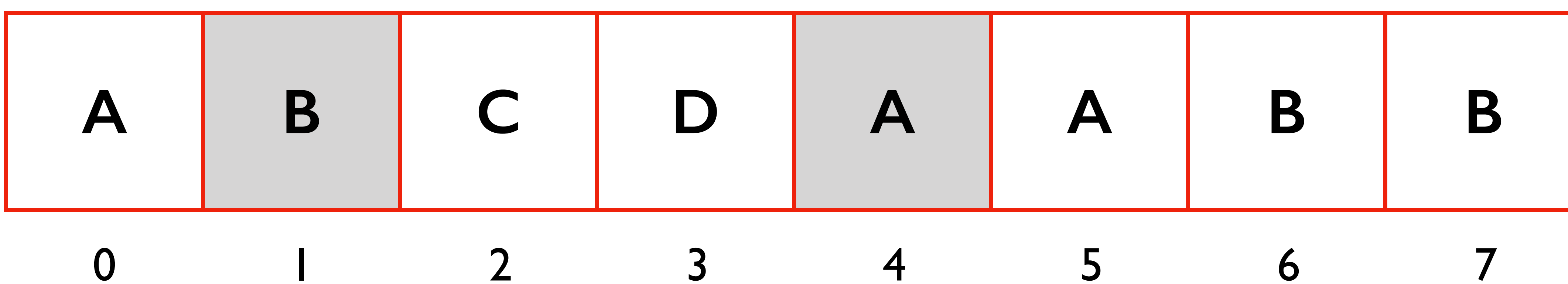
Reading Data

Compacting Data

Buffering Writes

writes: block[1] = X, block[4] = B

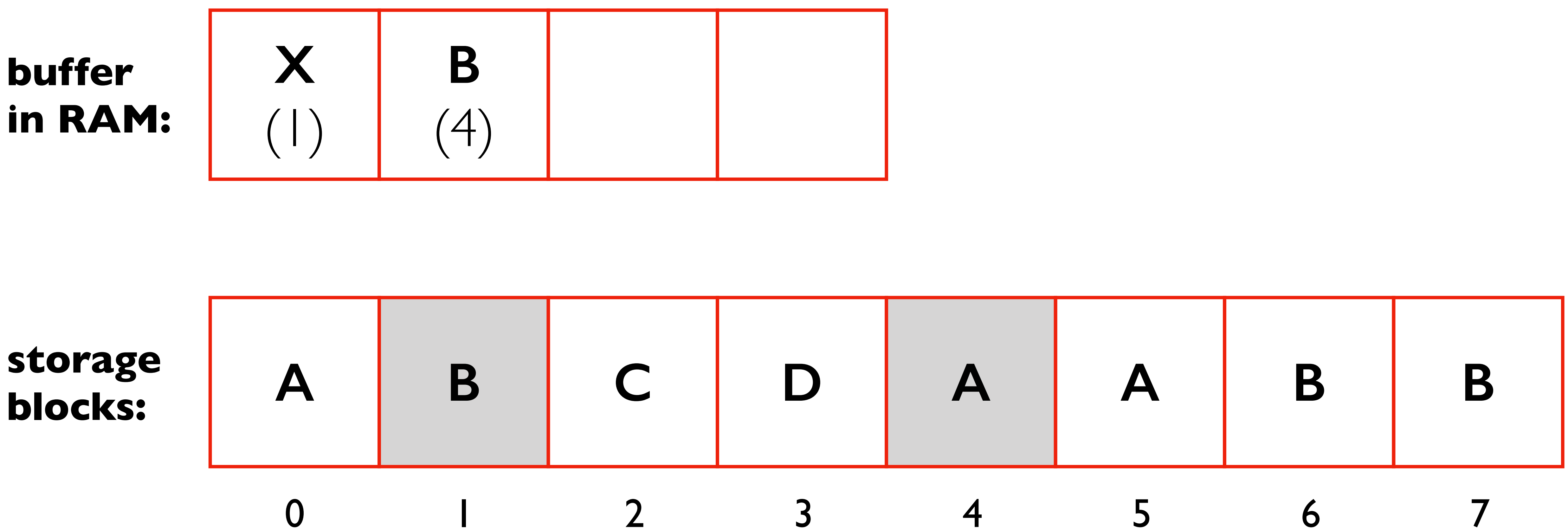
storage blocks:



waiting for the write would be slow:
storage is slow in general
(these writes would be random)

Buffering Writes

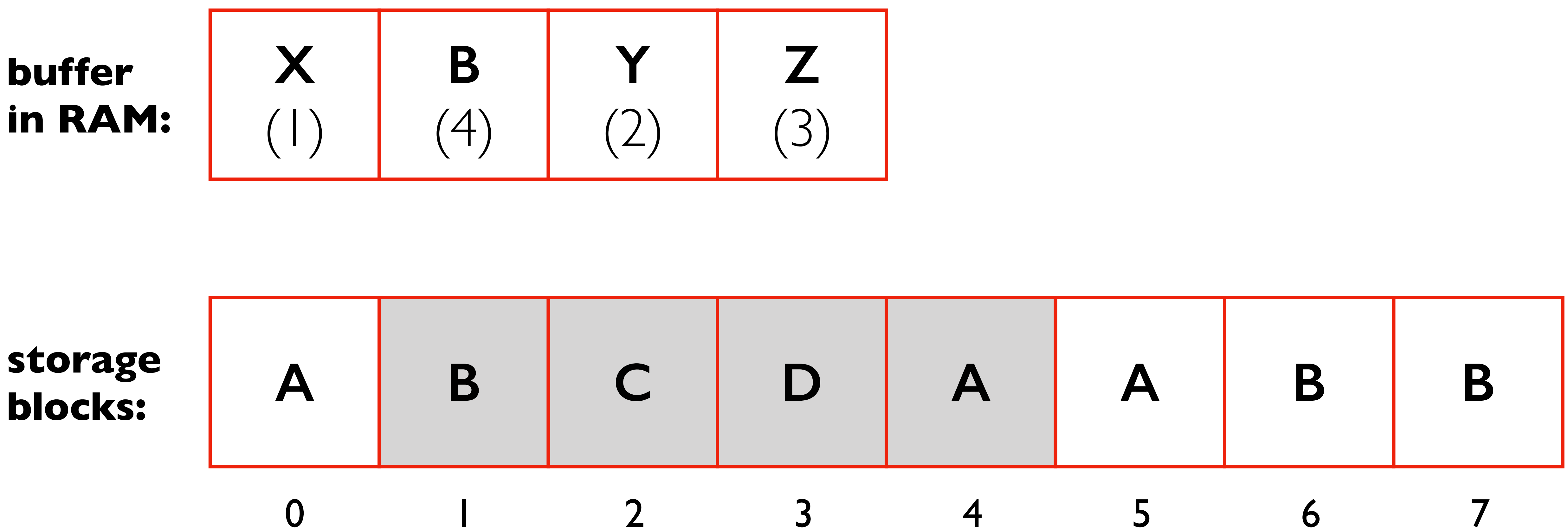
writes: block[1] = X, block[4] = B



return from write now, buffer work for later...

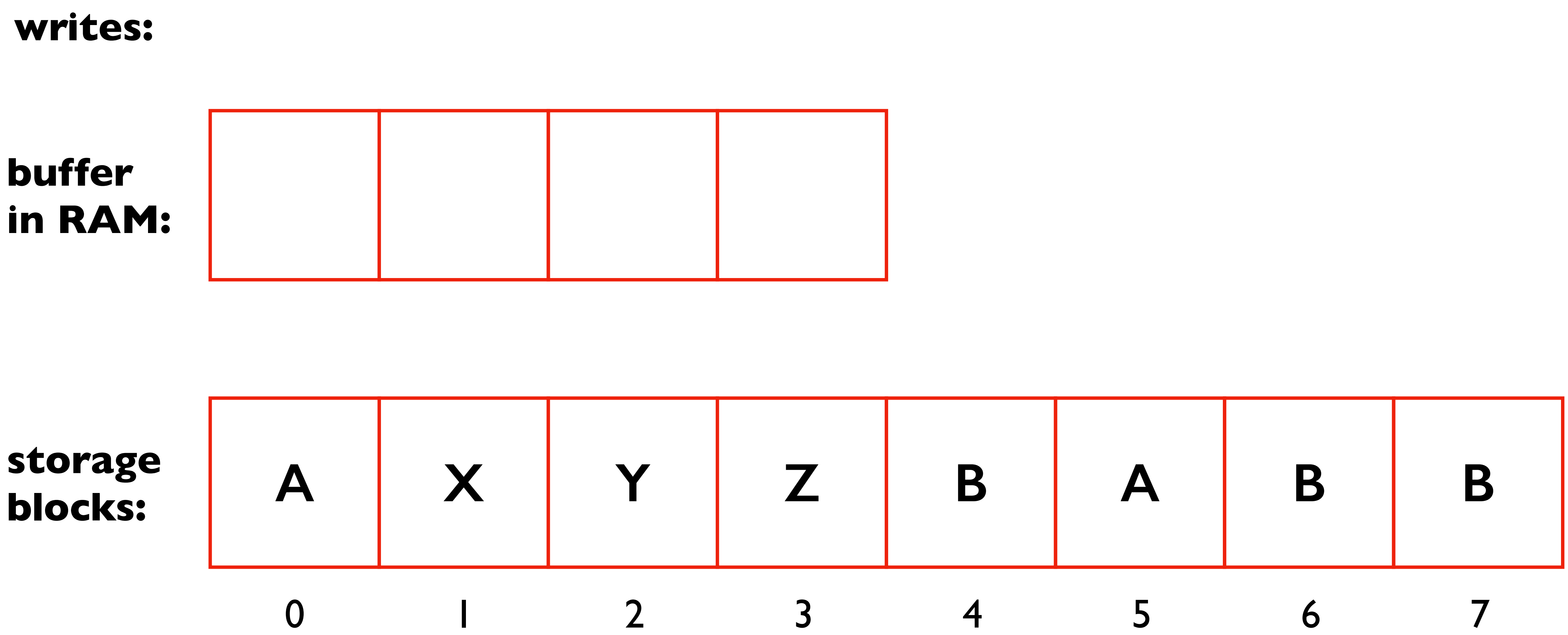
Buffering Writes

writes: block[1] = X, block[4] = B, block[2] = Y, block[3] = Z



if we're lucky, we'll get more writes that we can do efficiently together

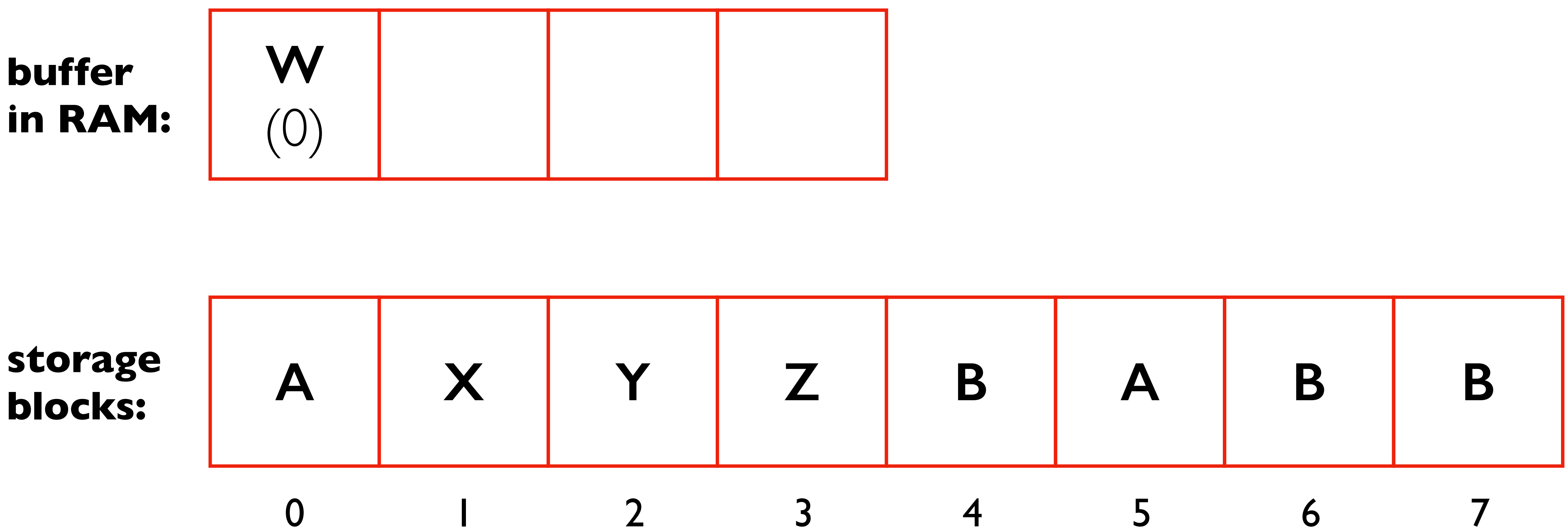
Buffering Writes



sync to disk eventually

Crashing at a bad time

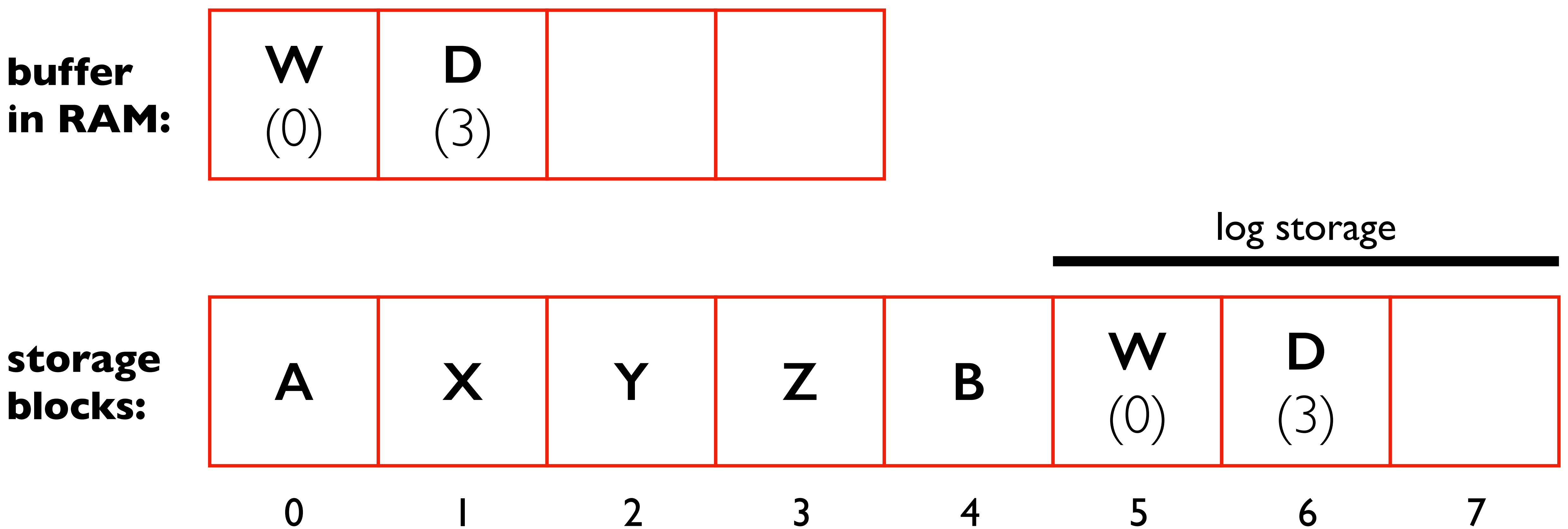
writes: block[0] = W



what will happen to the last written data?

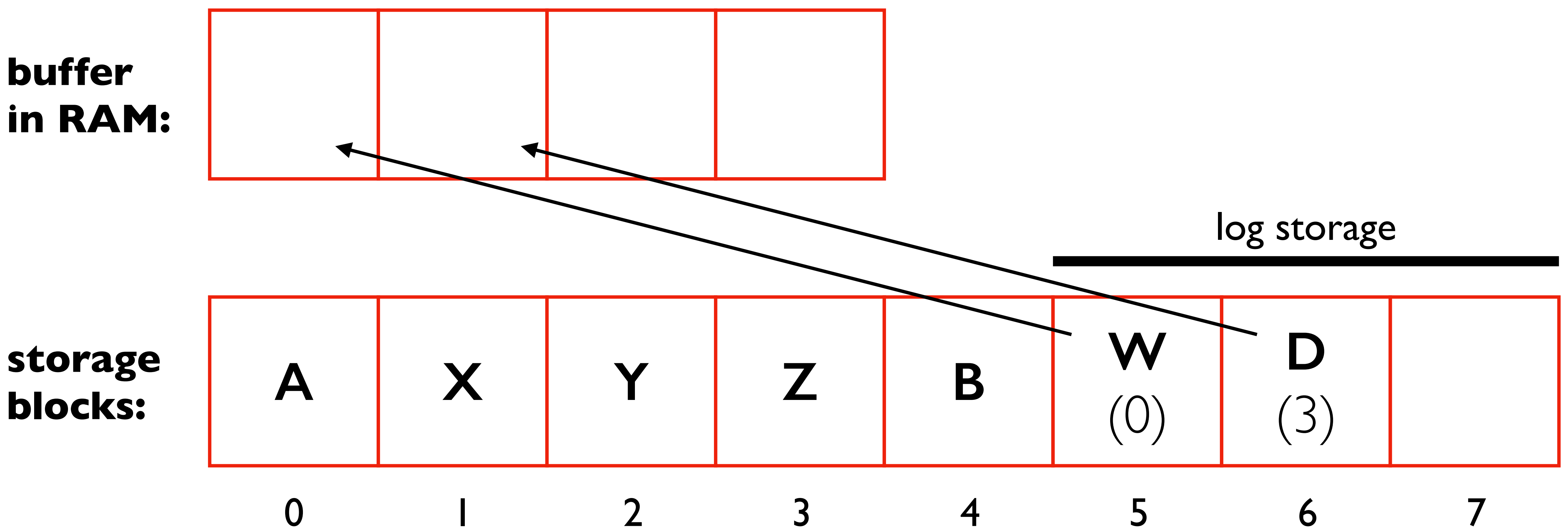
Logging Writes

writes: block[0] = W, block[3]=D



sequentially write it to the log now; write to the correct place later

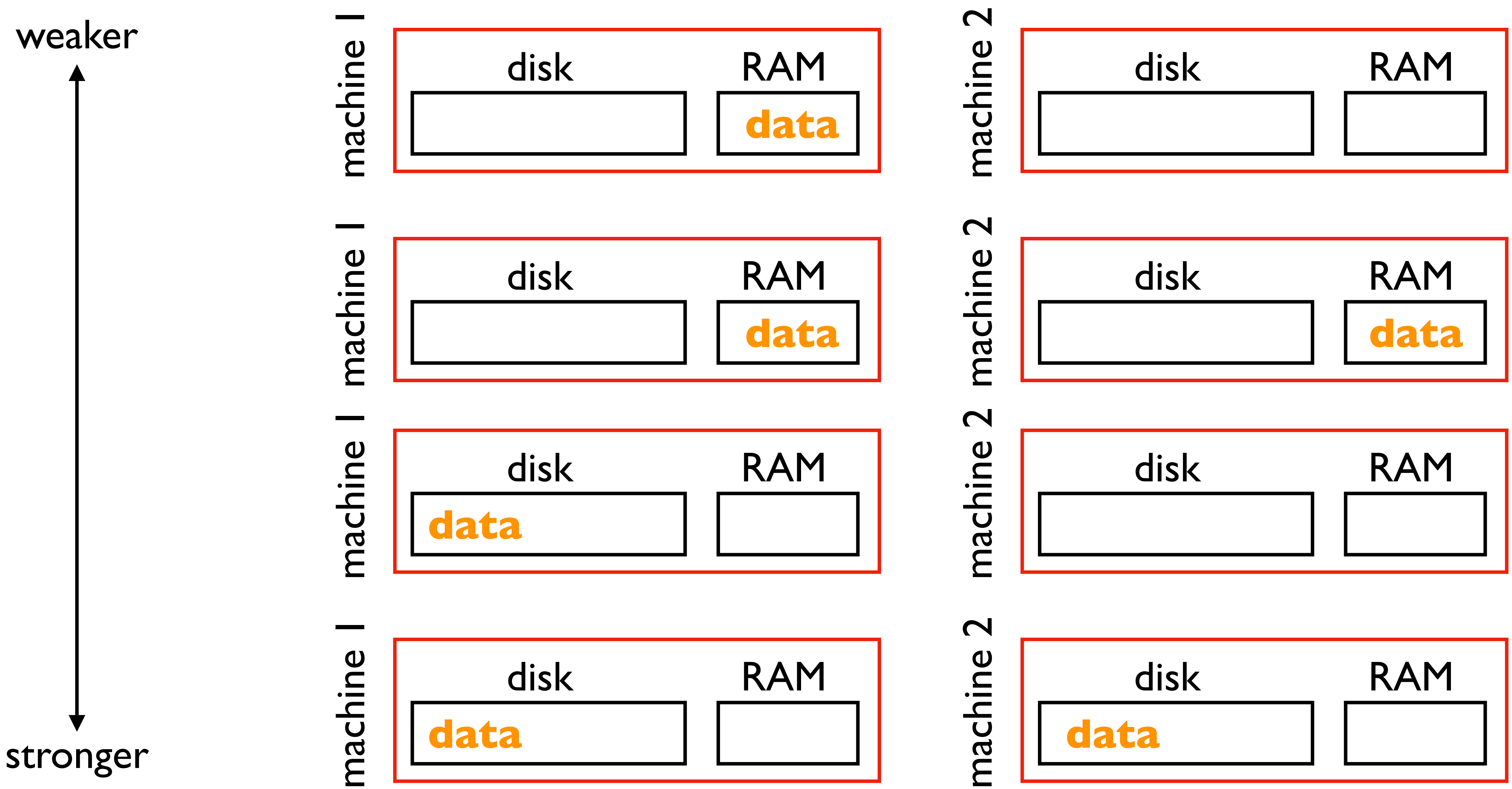
Recovery



normally a log is never read, just after restarting from a crash

Durability in a Distributed System

Durability means your data isn't lost when certain bad things happen. Stronger durability means tolerance for more kinds of faults.



Outline: Cassandra Storage Engine

Writing Data: Buffering and Logging

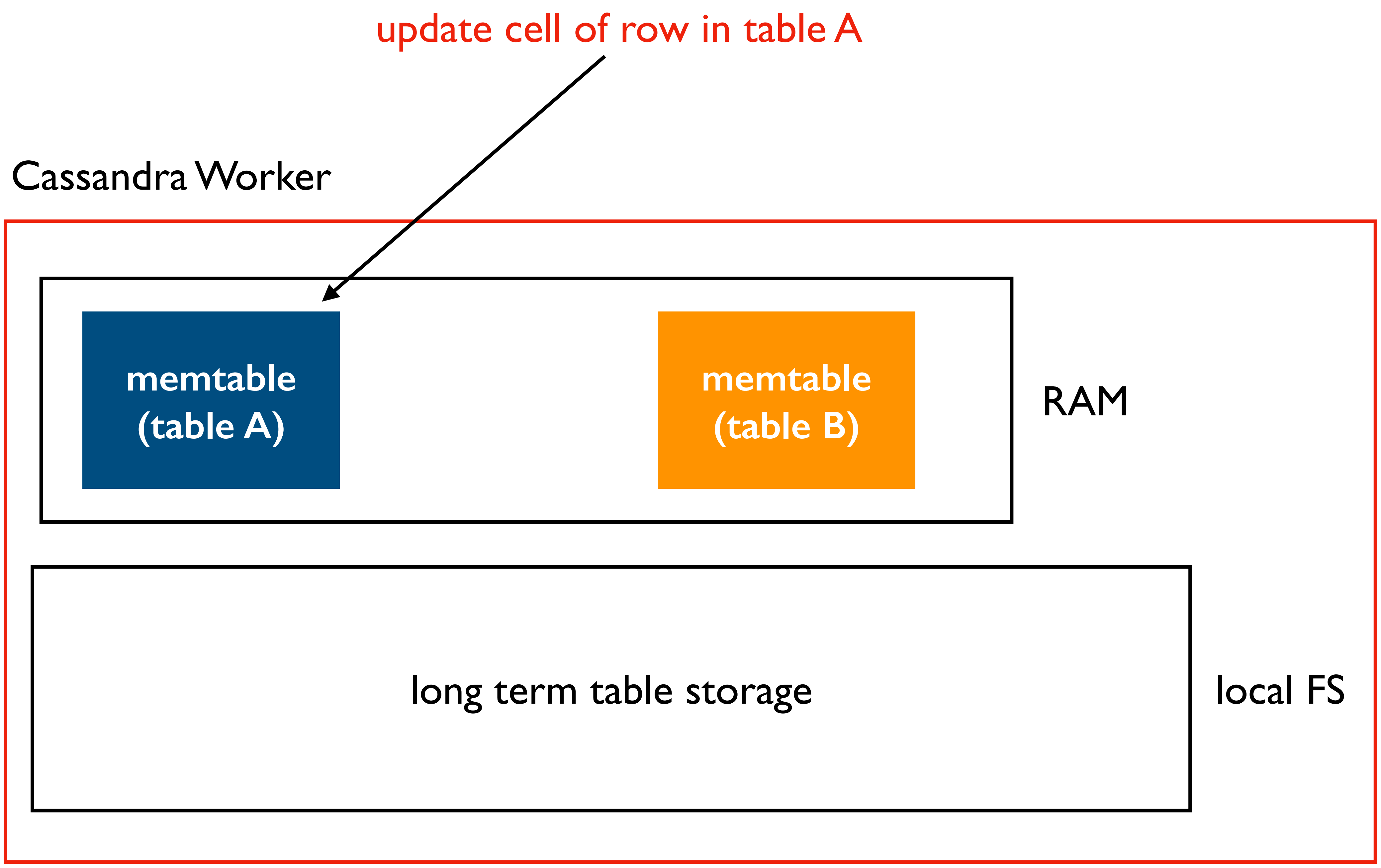
- in general
- Cassandra

Storing Data: SSTables

Reading Data

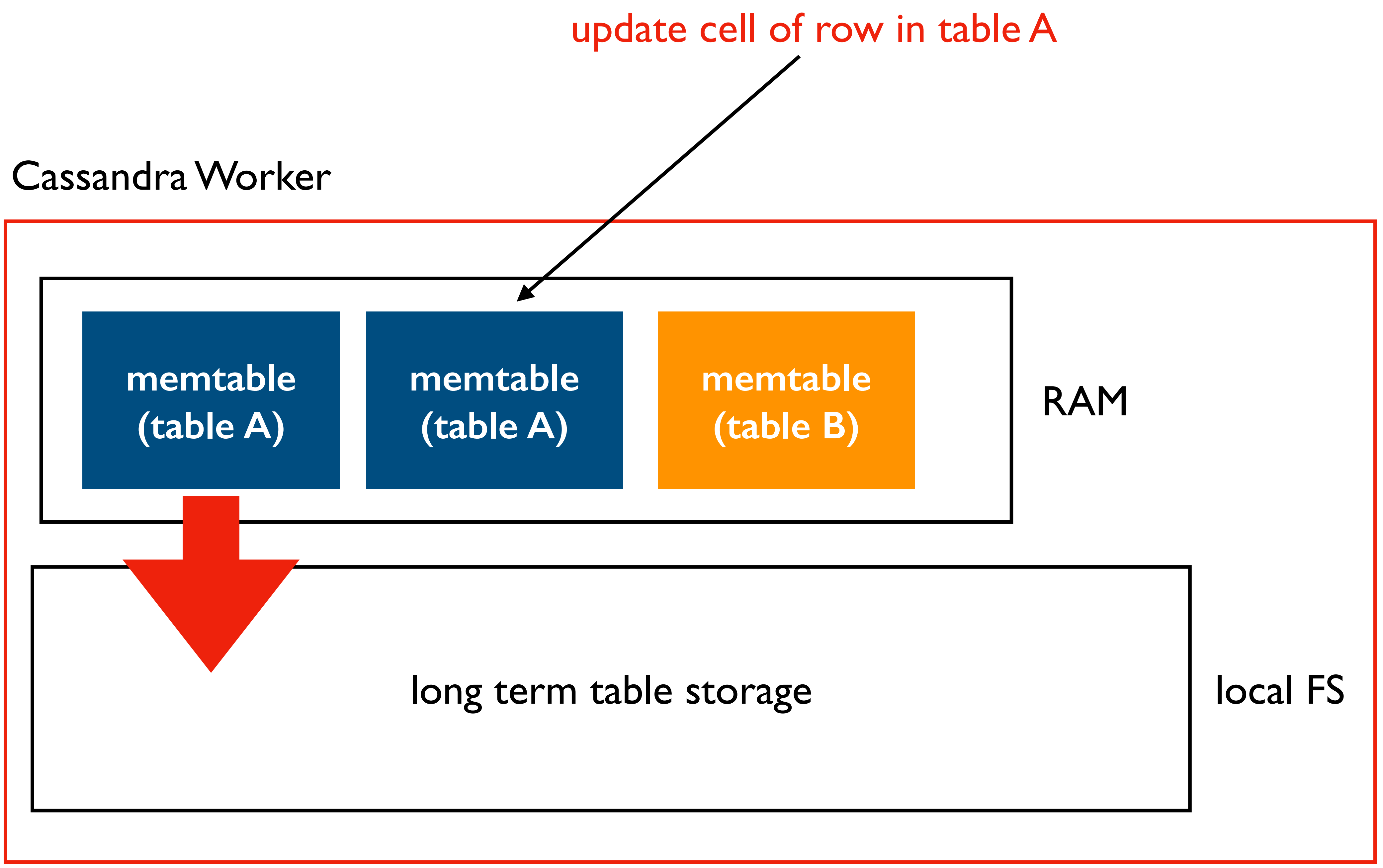
Compacting Data

Memtables



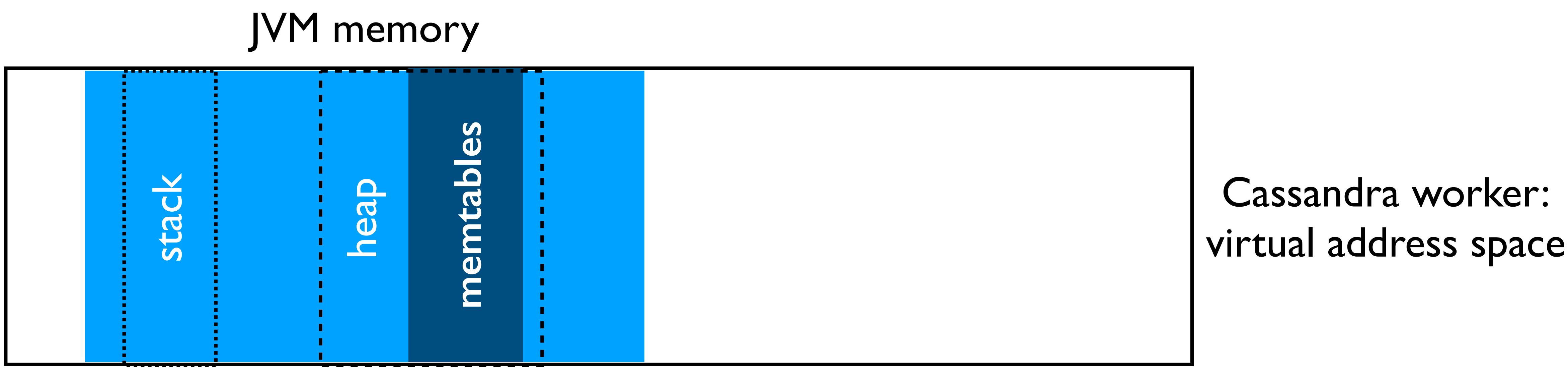
each Cassandra table will have its own in-memory memtable

Memtables



occasionally there are multiple memtables for the same table
(one can be flushed to disk while another receives new writes)

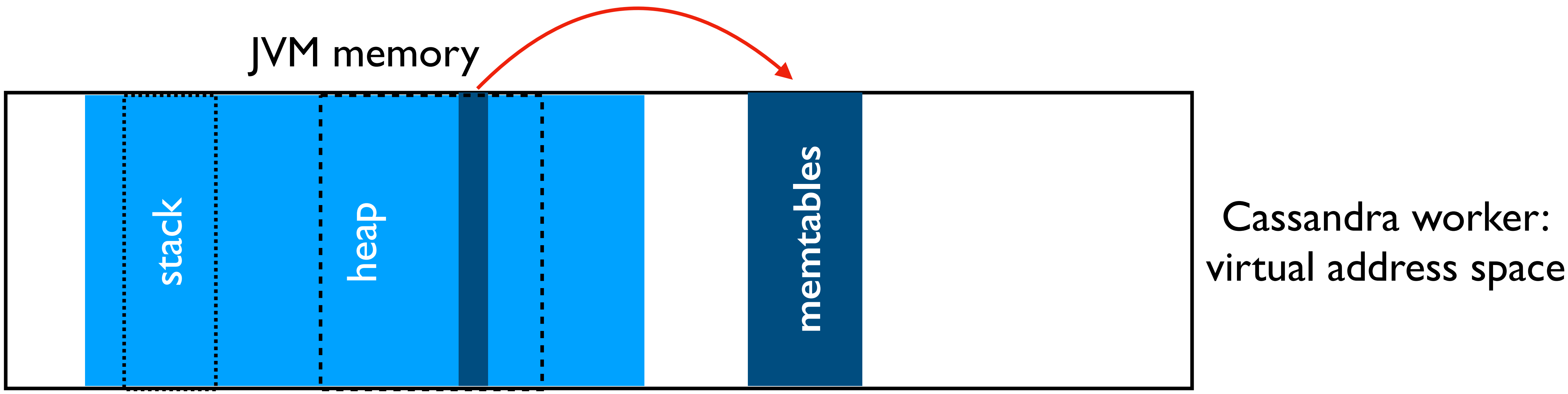
Memtables, Memory Layout



remember, JVM memory management is notoriously inneficient and prone to garbage collection pauses

https://www.datastax.com/blog/heap-memtables-cassandra-21

Memtables, Memory Layout

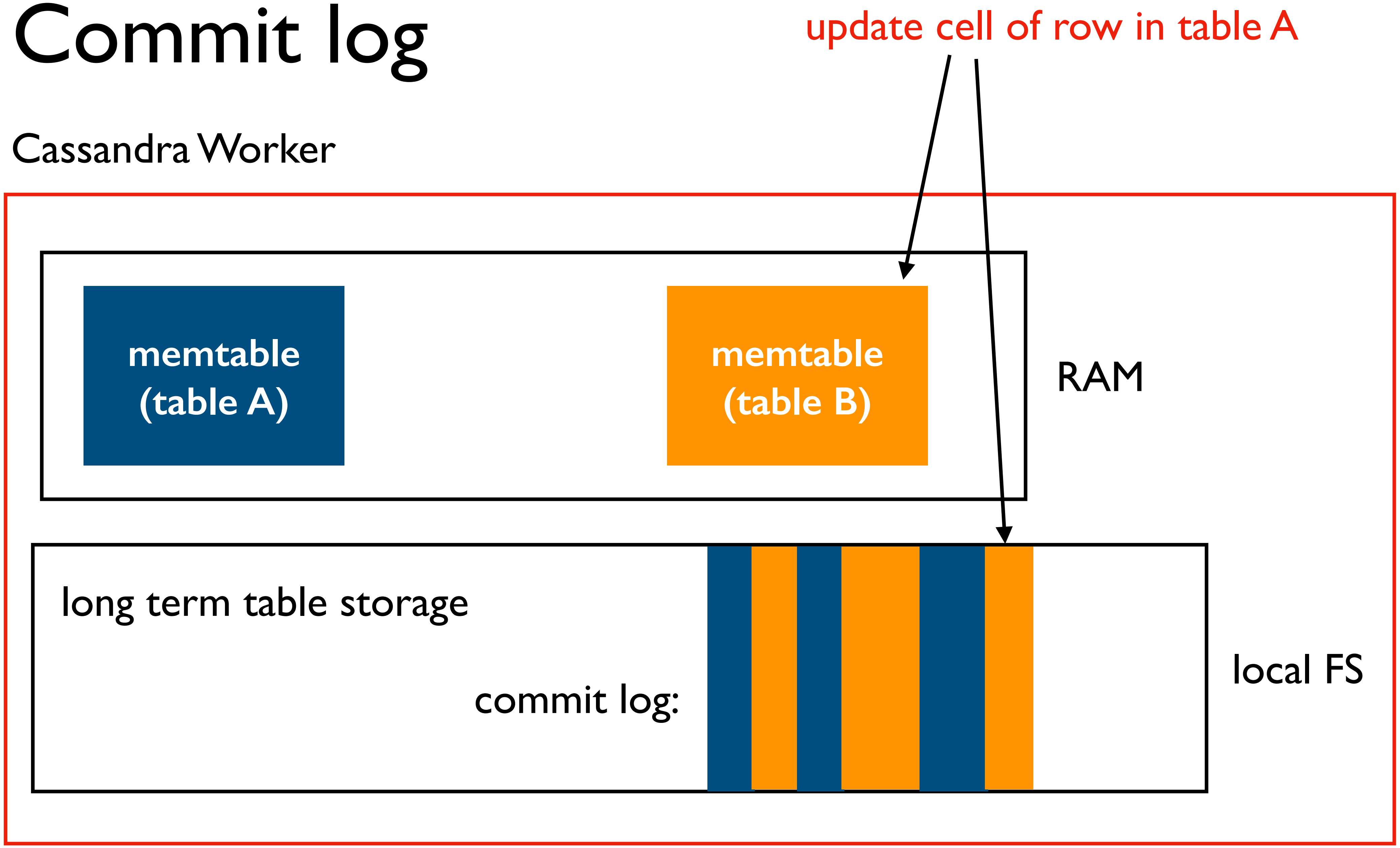


remember, JVM memory management is notoriously inneficient and prone to garbage collection pauses

latest Cassandra versions can store most memtable data off heap

https://www.datastax.com/blog/heap-memtables-cassandra-21

Commit log



a single log per worker is shared between all tables and written sequentially

keyspaces can be tuned for either performance (log off) or durability (log on)

CREATE KEYSPACE ???? WITH REPLICATION={...} AND DURABLE_WRITES=true

Outline: Cassandra Storage Engine

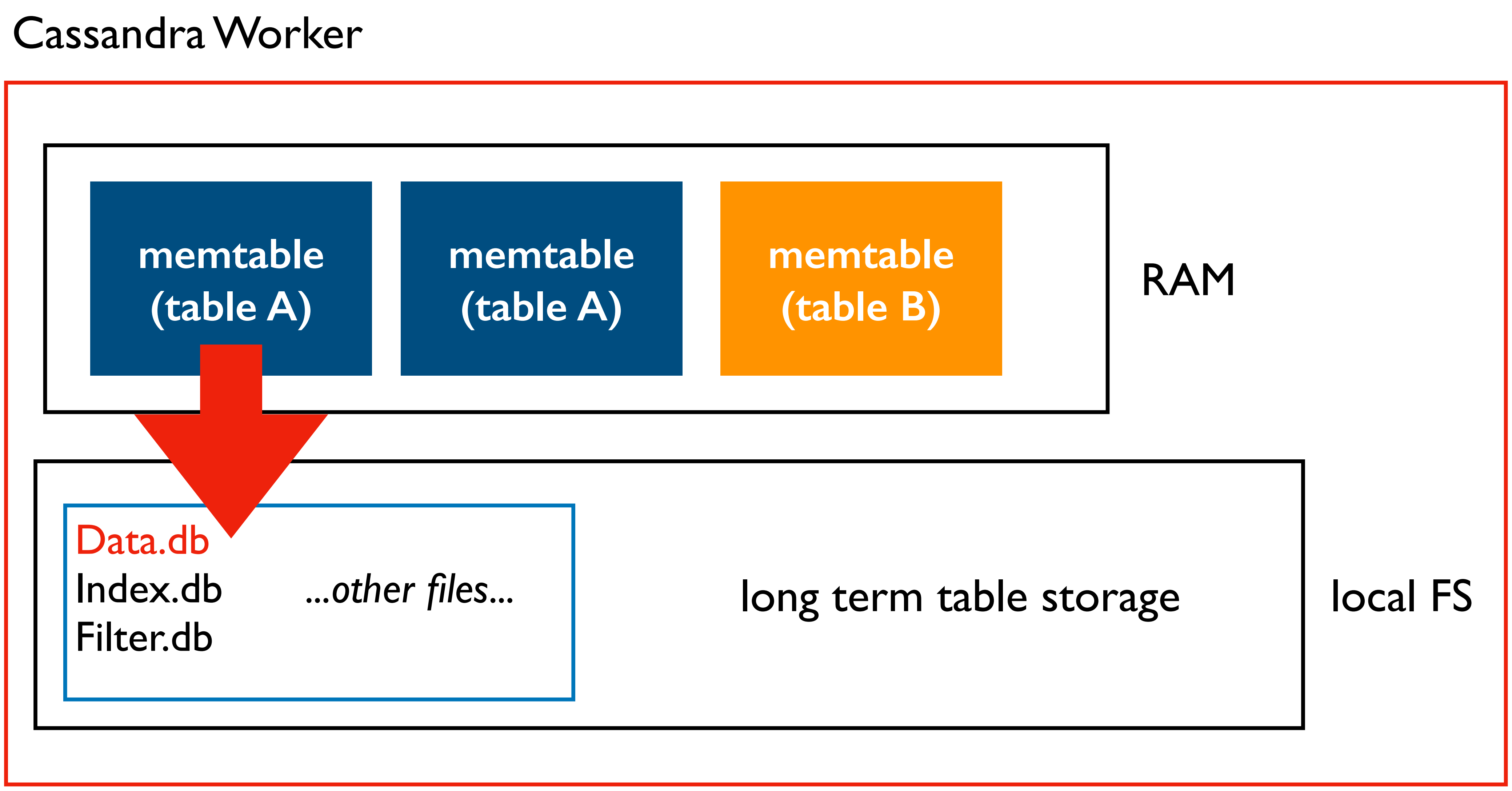
Writing Data: Buffering and Logging

Storing Data: SSTables

Reading Data

Compacting Data

SSTables (Sorted String Tables)



it is sorted because Data.db contains key/value pairs sorted by key
(which actually is not required to be a string)

Data.db

key	value
A	(1,4)
C	(9,3)
D	(2,8)
G	(5,4)
M	*tombstone*
Q	(2,4)

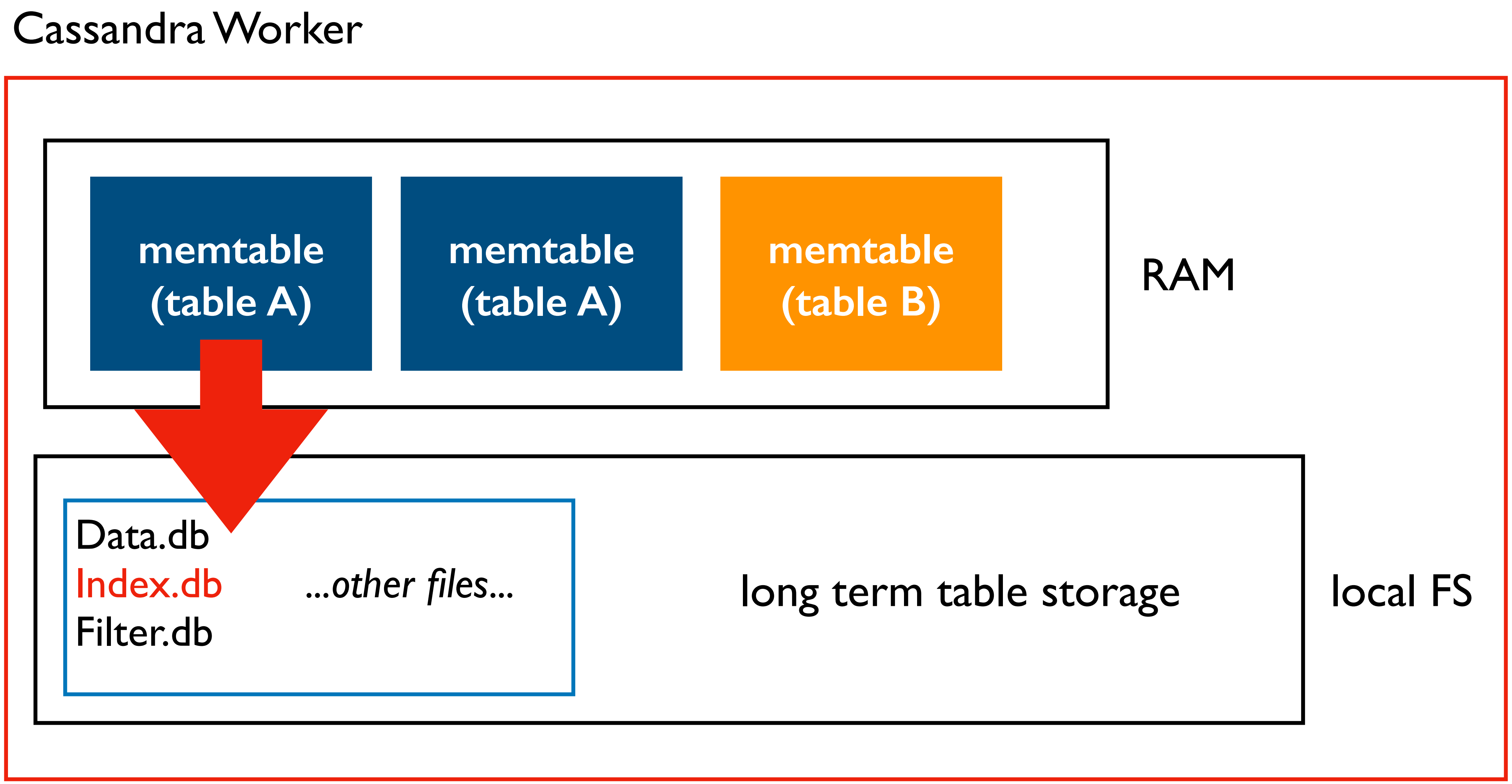
sorting makes "range queries" (lookups of consecutive keys) efficient/sequential

indicates deletion

row data stored here

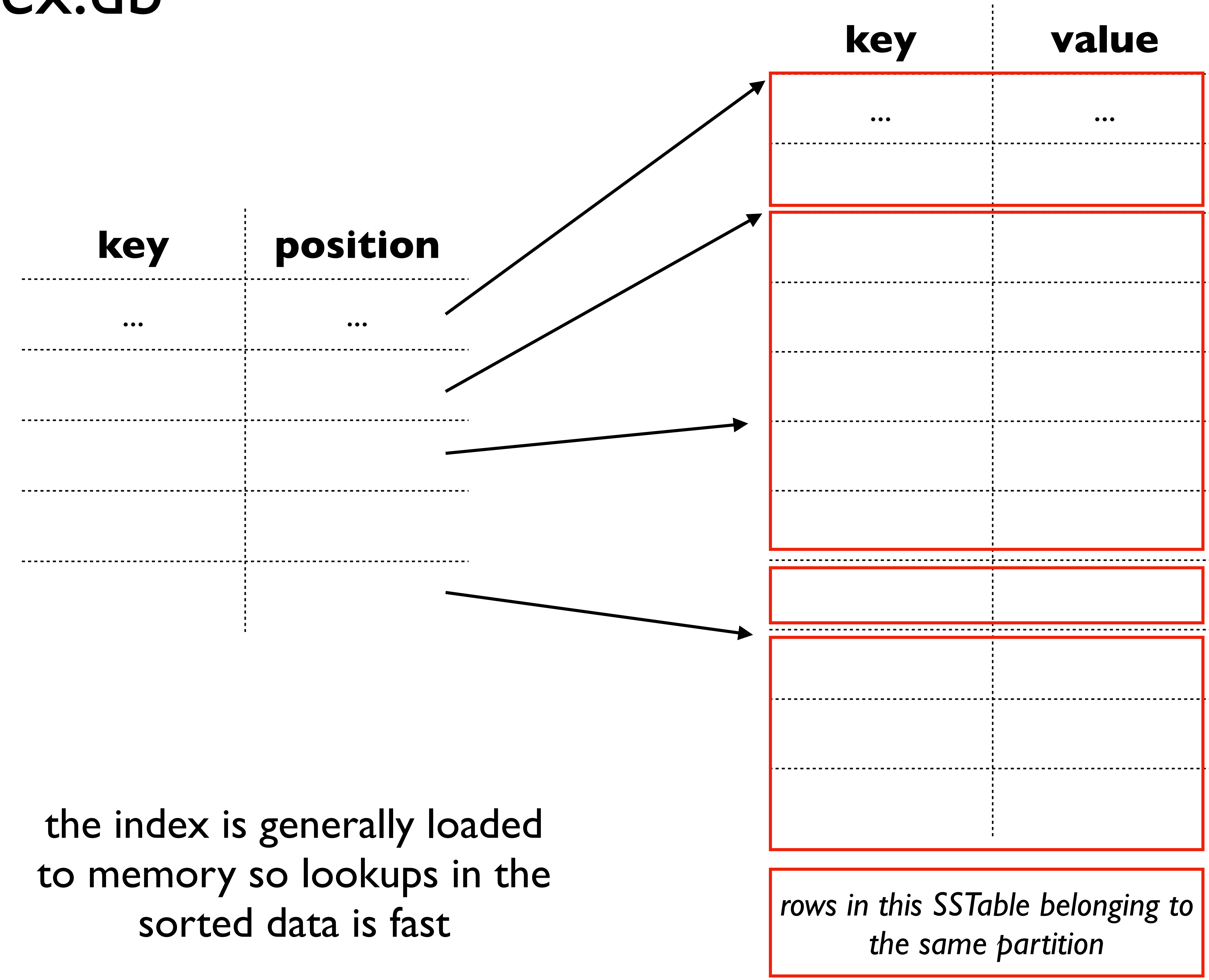
this key will encode primary keys (partition+cluster keys)

SSTables (Sorted String Tables)



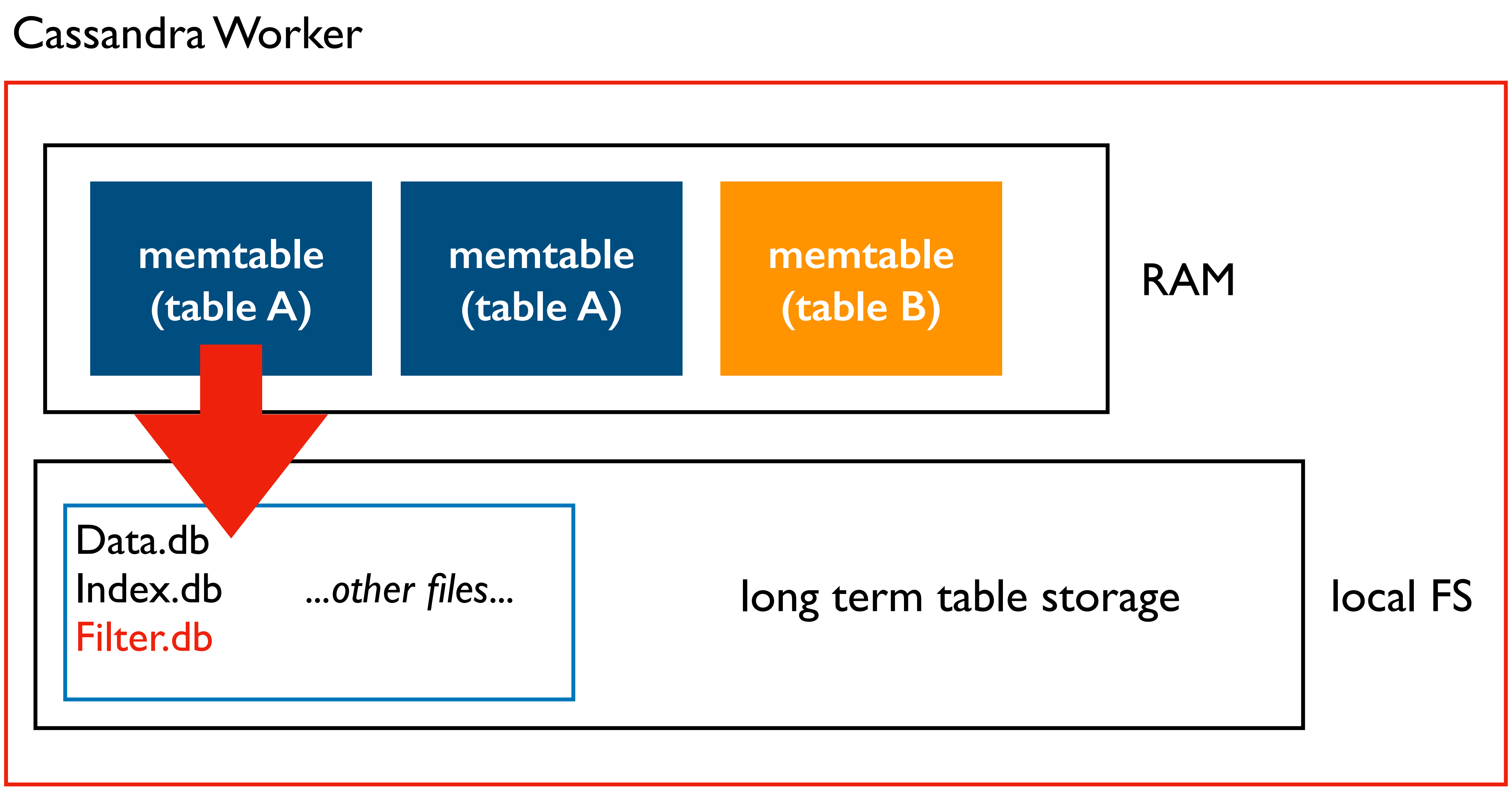
problem: even though Data.db is sorted, we wouldn't want to do binary search on it to find an entry, because the disk I/O would be random

Index.db



the index is generally loaded to memory so lookups in the sorted data is fast

SSTables (Sorted String Tables)



can we guess whether a key is in the data without actually looking?

Filter.db contains a bloom filter, a very space efficient structure for helping with this. Like Index.db, it is generally loaded to memory.

Filter.db: bloom filter construction

Step 1: compute two different hash functions for every key (mod N)

key	hash1(key)%N	hash2(key)%N
A	2	8
B	4	8
...		

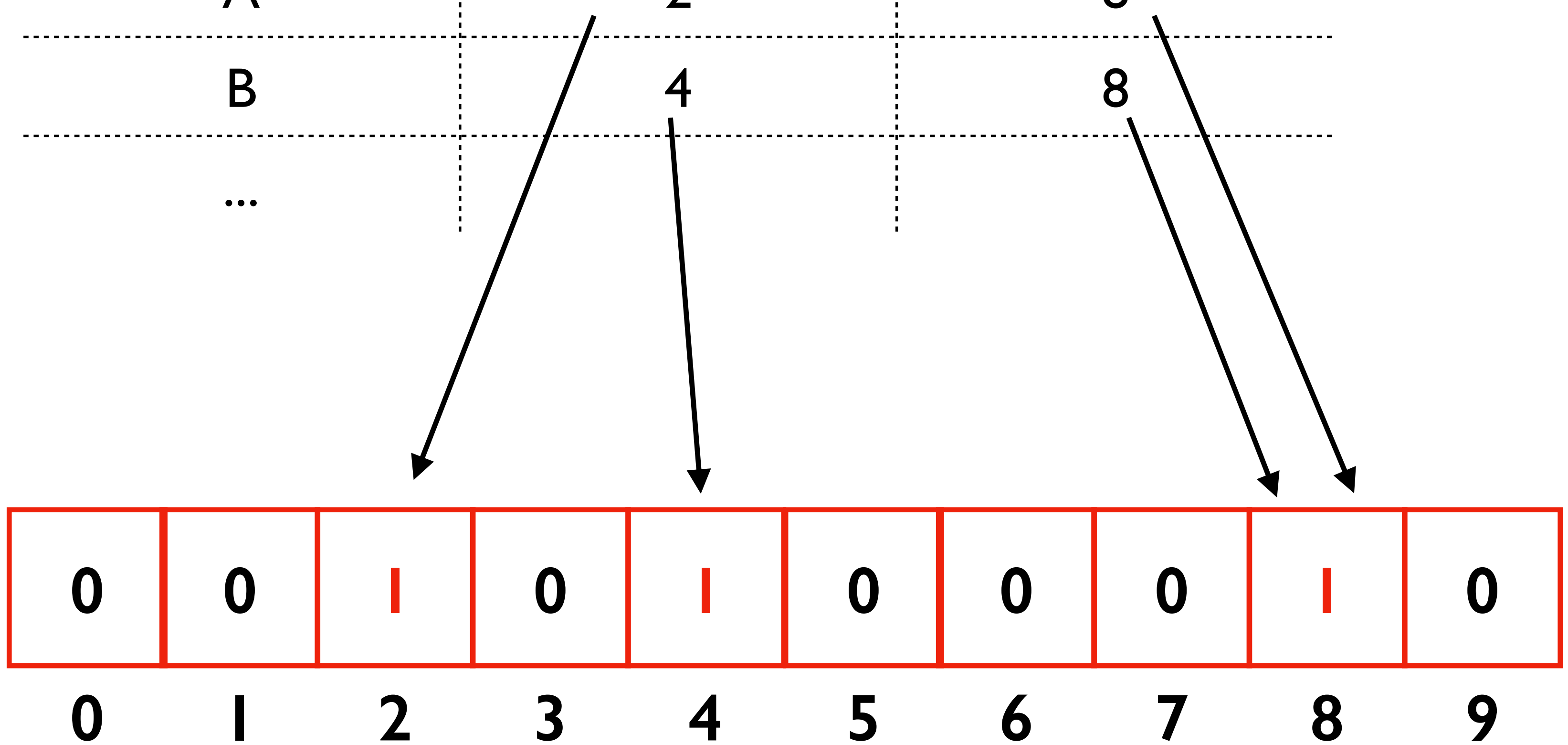
0	0	0	0	0	0	0	0	0	0
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**

N=10 bits of bloom filter

Filter.db: bloom filter construction

Step 1: compute two different hash functions for every key (mod N)

key	hash1(key)%N	hash2(key)%N
A	2	8
B	4	8
...		



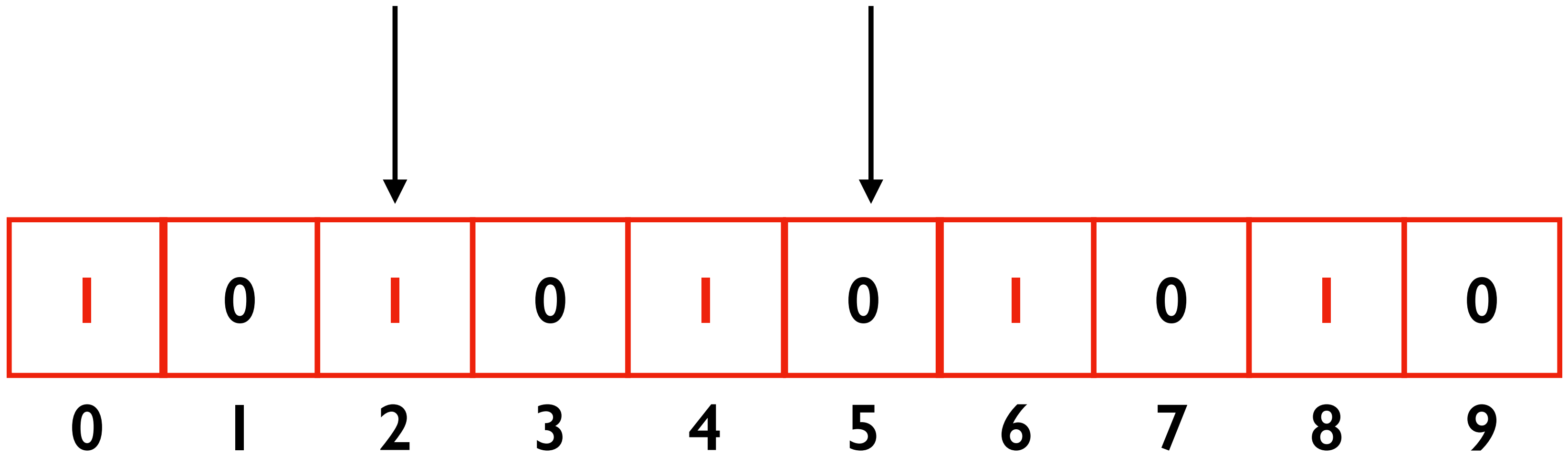
N=10 bits of bloom filter

Step 2: flip zeros to ones at each position corresponding to a hash%N

Filter.db: bloom filter lookup: no case

Was C inserted in the bloom filter?

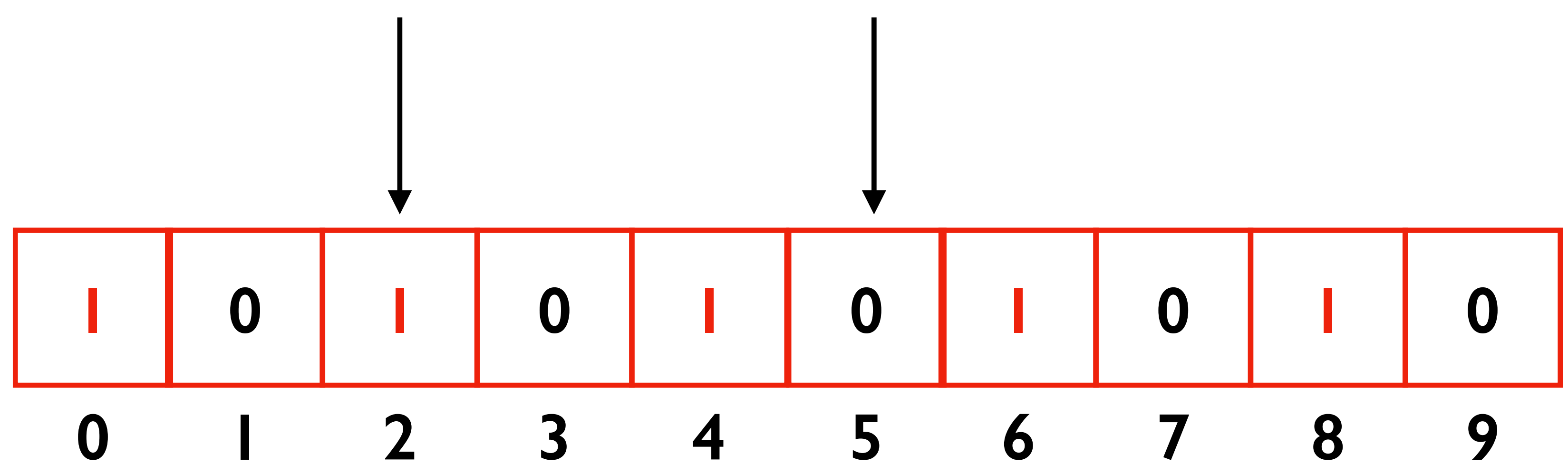
Assume hash1(C) % N = 2
AND hash2(C) % N = 5



Filter.db: bloom filter lookup: no case

Was C inserted in the bloom filter?

Assume hash1(C) % N = 2
AND hash2(C) % N = 5

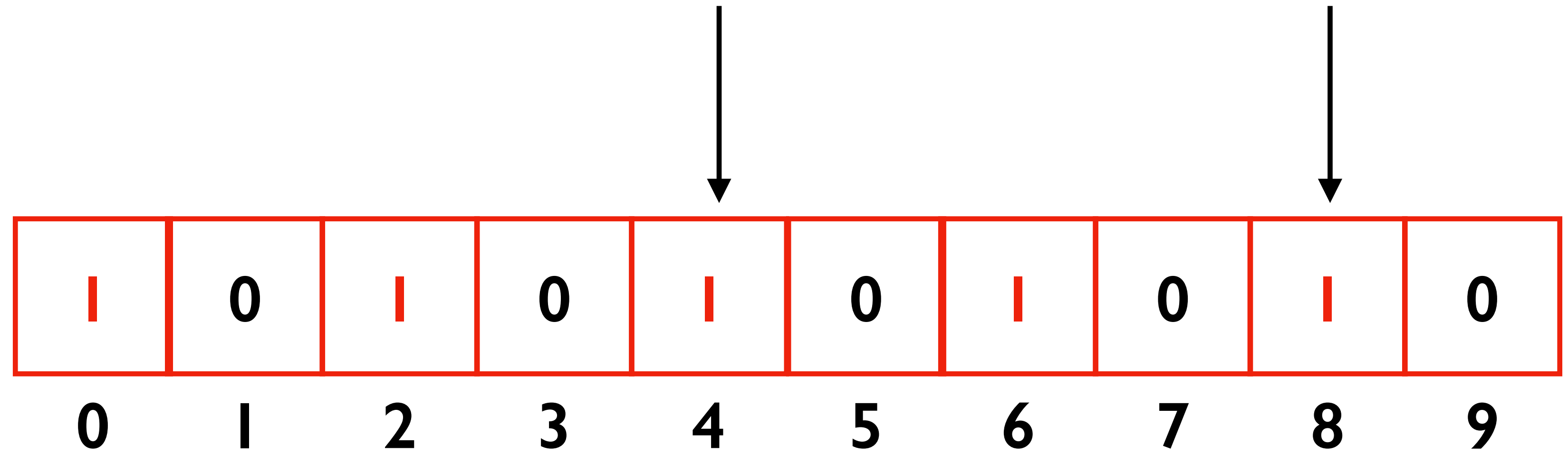


It definitely was NOT inserted. Otherwise we would have flipped position 5 to a one.

Filter.db: bloom filter lookup: maybe case

Was D inserted in the bloom filter?

Assume hash1(D) % N = 4
AND hash2(D) % N = 8



Filter.db: bloom filter lookup: maybe case

Was D inserted in the bloom filter?

Assume hash1(D) % N = 4
AND hash2(D) % N = 8

1	0	1	0	1	0	1	0	1	0
0	1	2	3	4	5	6	7	8	9

Maybe it was, as both spots are 1's. Or it could be a false positive (remember that A and B together flipped these positions).

Bloom filters

False positive rate depends on

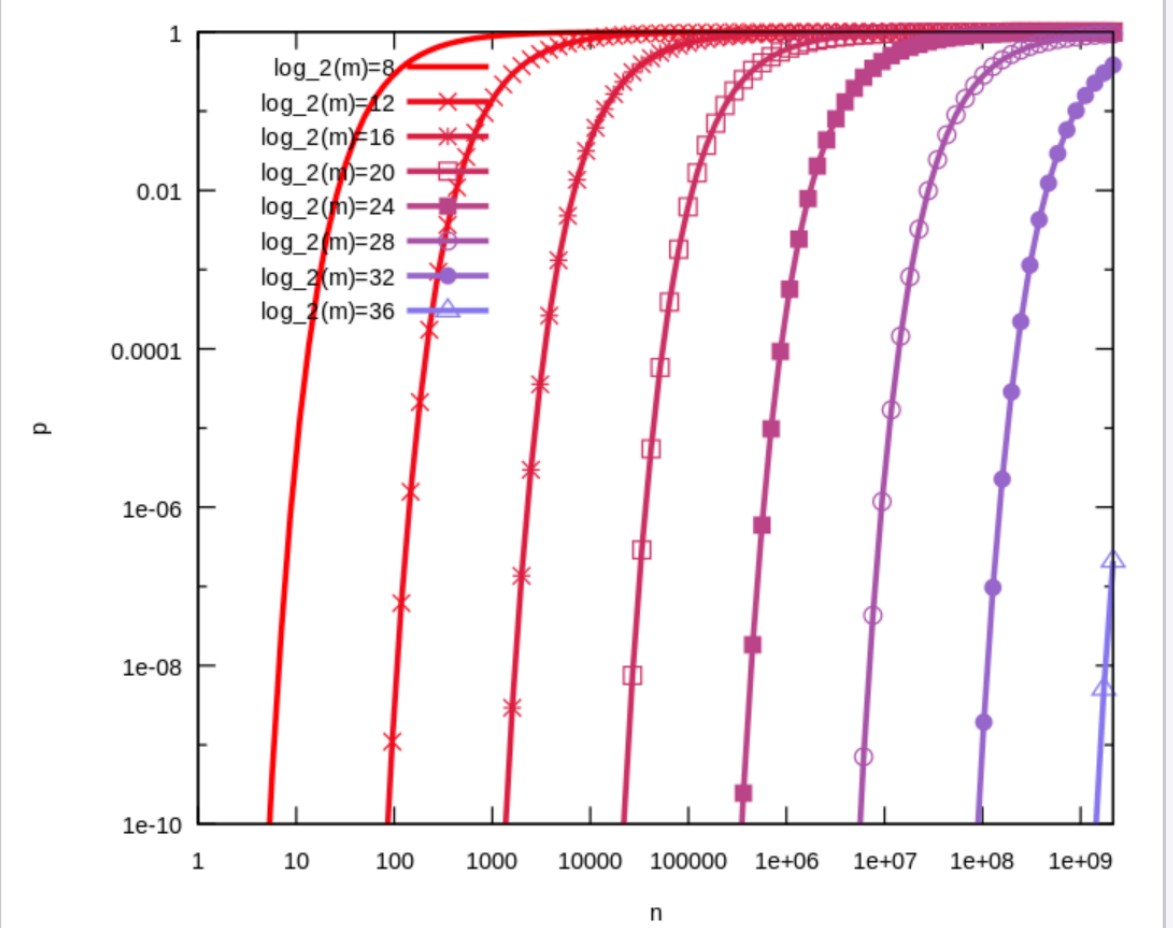
- number of inserts
- number of bits
- number of hash functions

It can be tuned to achieve any desirable rate (but a lower rate means more memory).

```
# if it were a Python
# data structure:
my_set = {...}
my_bloom = ...

x in my_set   # True or False
x in my_bloom # Maybe or False (NEVER True)
```

https://en.wikipedia.org/wiki/Bloom_filter



The false positive probability p as a function of number of elements n in the filter and the filter size m. An optimal number of hash functions $k = (m/n)\ln 2$ has been assumed.

TopHat

Outline: Cassandra Storage Engine

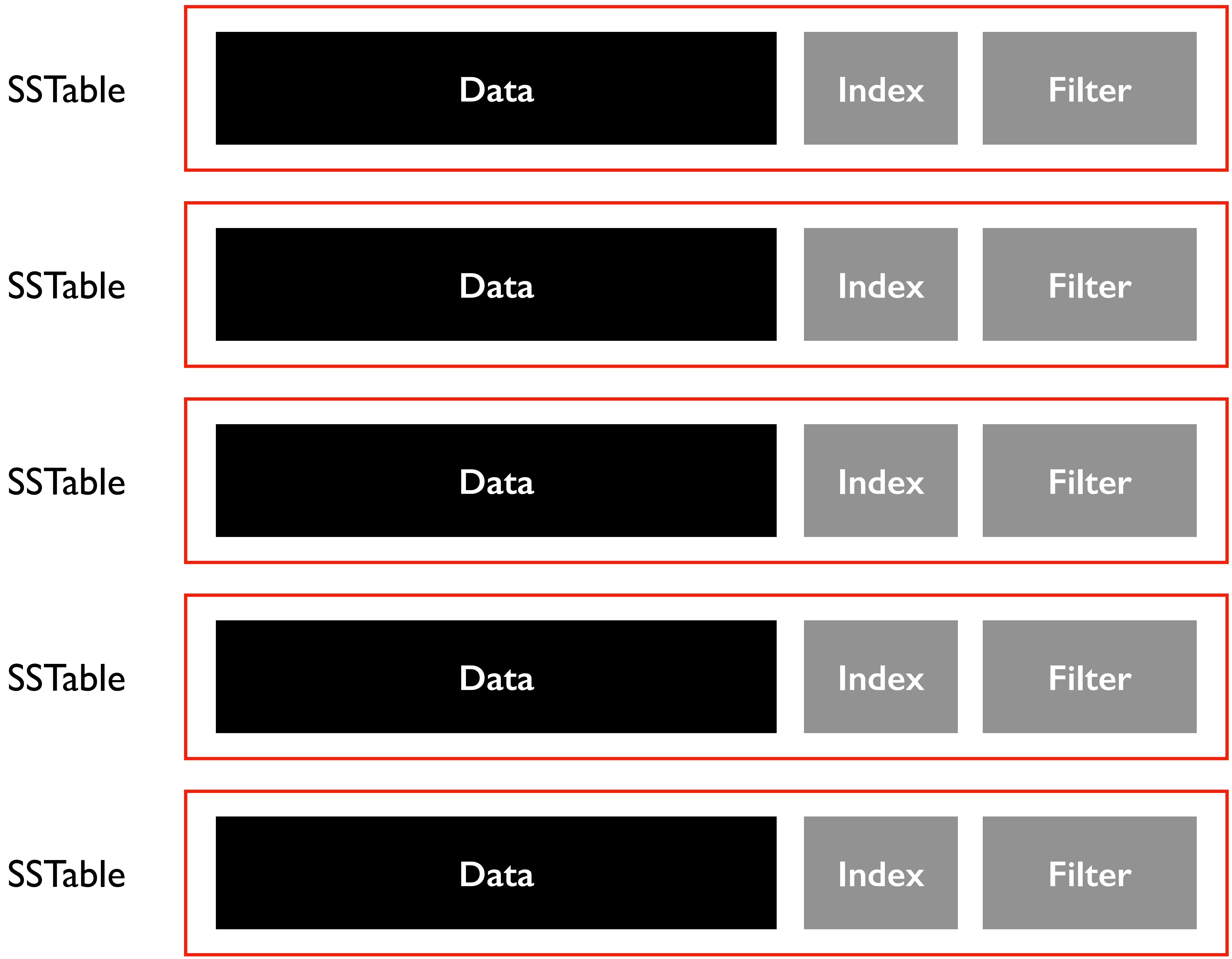
Writing Data: Buffering and Logging

Storing Data: SSTables

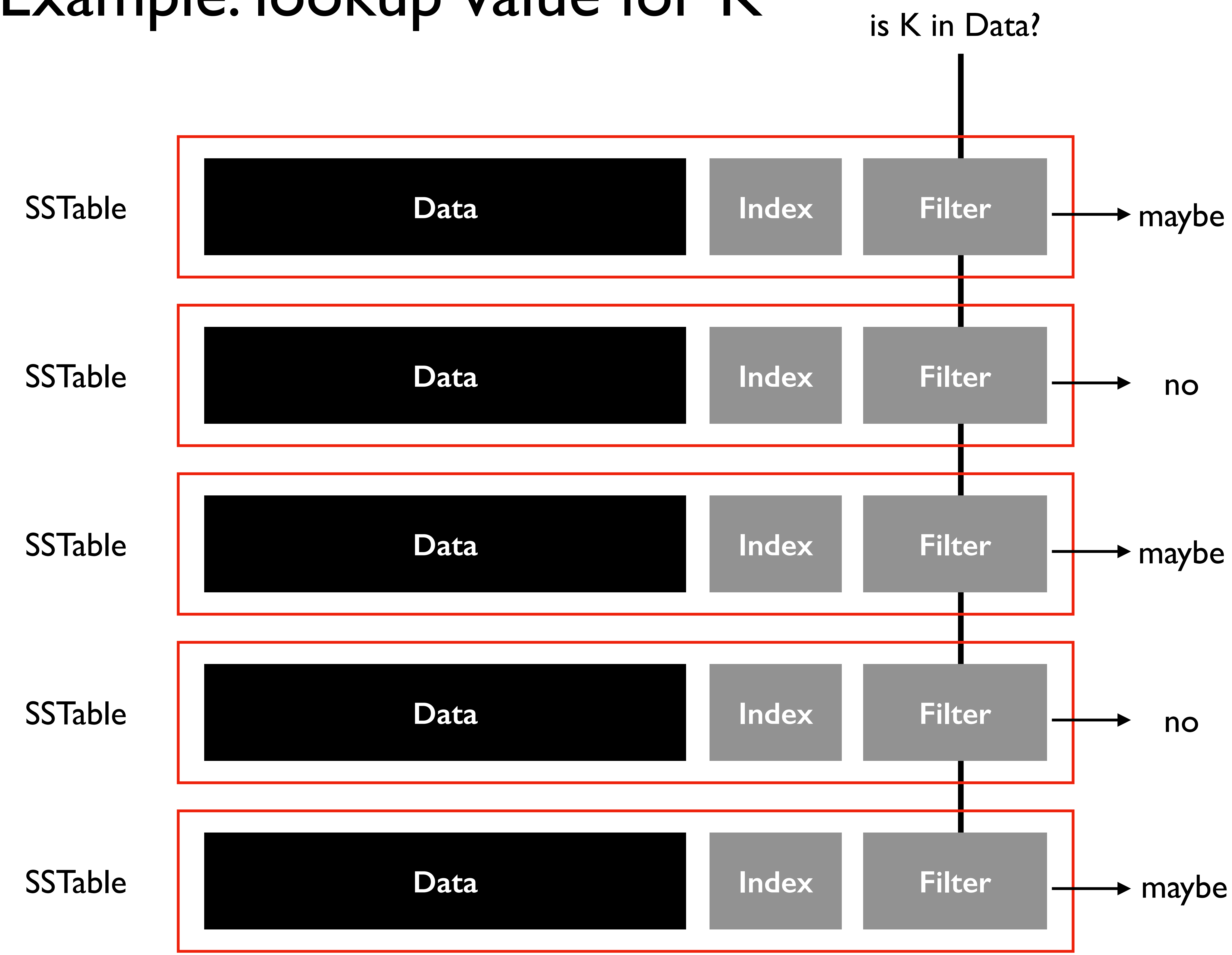
Reading Data

Compacting Data

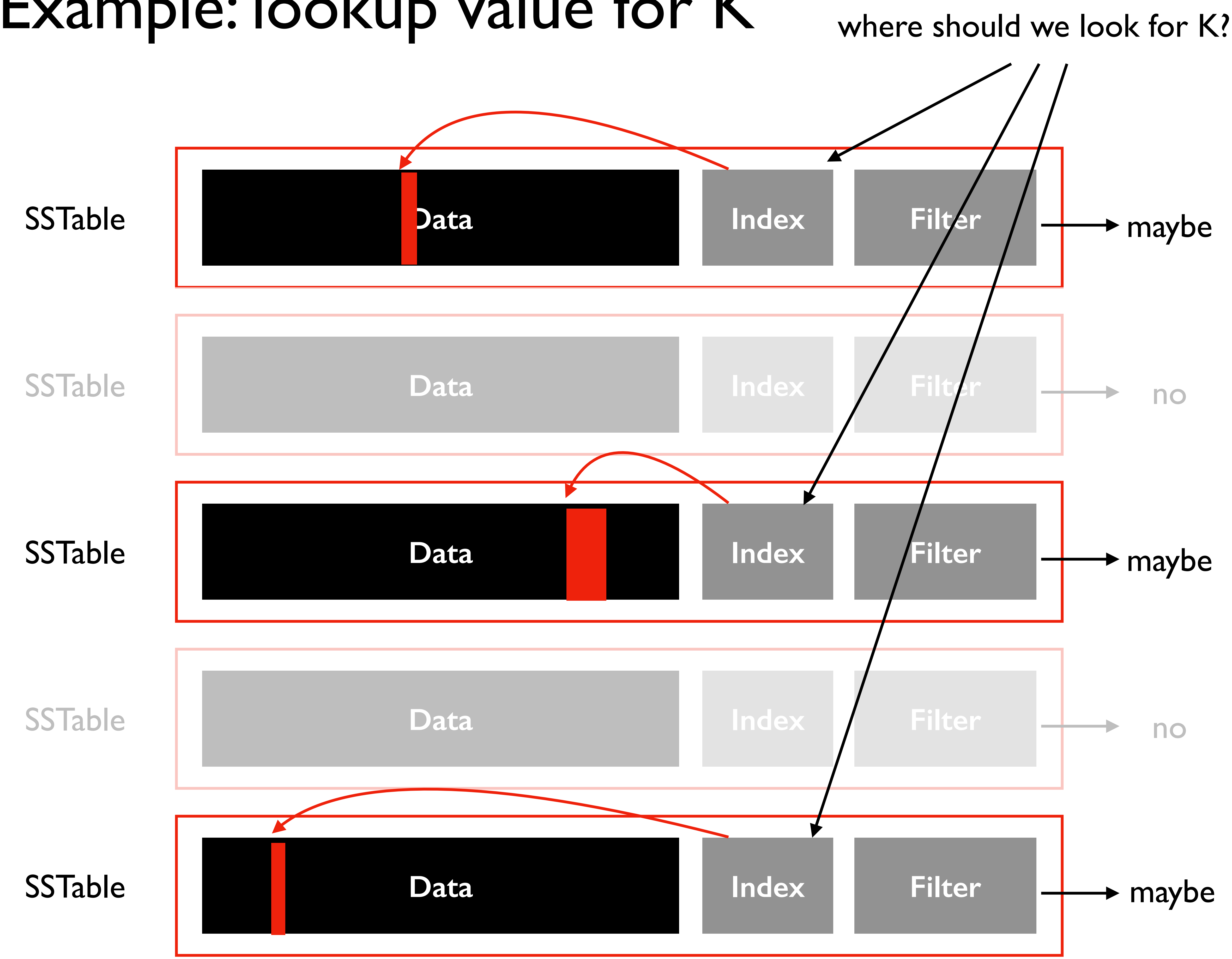
Example: lookup value for K



Example: lookup value for K



Example: lookup value for K



Example: lookup value for K

lookup[K] = v2

Outline: Cassandra Storage Engine

Writing Data: Buffering and Logging

Storing Data: SSTables

Reading Data

Compacting Data

- merge sort
- performance considerations
- HBase vs. Cassadra

Controlling Read Cost

Only dumping out large, immutable SSTables is good for writes (everything is sequential).

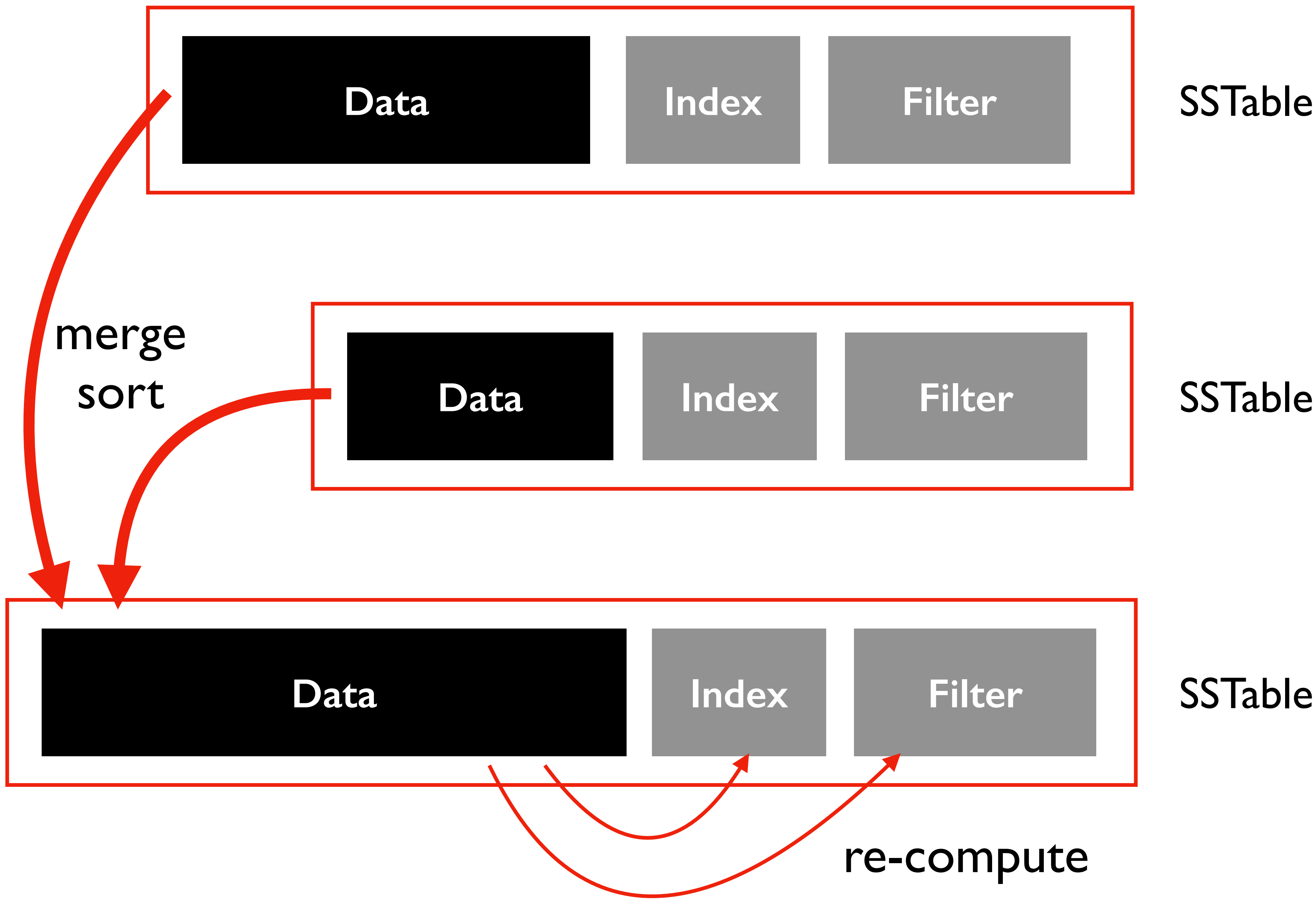
Over time, read cost grows with number of SSTables.

Bloom filters help, but have limitations:

- false positives
- same key might be any many SSTables
- sometimes want to read values between K1 and K2 (a "range query") instead of for a single K value (a "point lookup")

We can reduce the number SSTables by "merge sorting" multiple small ones into one bigger one. This is called "compaction".

Compaction: Merge Sorting SSTables



Compaction

newer

key	value
A	(1,4)
G	(8,9)
M	*tombstone*

older

key	value
C	(9,3)
D	(2,8)
G	(5,4)
M	(3,4)
Q	(2,4)

compacted

key	value

Compaction

newer

key	value
A	(1,4)
G	(8,9)
M	*tombstone*

older

key	value
C	(9,3)
D	(2,8)
G	(5,4)
M	(3,4)
Q	(2,4)

compacted

key	value
A	(1,4)

Compaction

newer

key	value
A	(1,4)
G	(8,9)
M	*tombstone*

older

key	value
C	(9,3)
D	(2,8)
G	(5,4)
M	(3,4)
Q	(2,4)

compacted

key	value
A	(1,4)
C	(9,3)
D	(2,8)

if both have same value, use newer

Compaction

newer

key	value
A	(1,4)
G	(8,9)
M	*tombstone*

older

key	value
C	(9,3)
D	(2,8)
G	(5,4)
M	(3,4)
Q	(2,4)

compacted

key	value
A	(1,4)
C	(9,3)
D	(2,8)
G	(8,9)

do we write a new tombstone, or delete the entry?

the key might appear in older SSTables: write tombstone

the key cannot appear in older SSTables: delete it

Compaction

newer

key	value
A	(1,4)
G	(8,9)
M	*tombstone*

older

key	value
C	(9,3)
D	(2,8)
G	(5,4)
M	(3,4)
Q	(2,4)

compacted

key	value
A	(1,4)
C	(9,3)
D	(2,8)
G	(8,9)
M	*tombstone*

once we get to the end of one input,
we just work from the others

Compaction

newer

key	value
A	(1,4)
G	(8,9)
	…bstone

older

key	value
C	…,3)
D	(2,8)
G	(5,4)
	(3,4)
Q	(2,4)

compacted

key	value
A	(1,4)
C	(9,3)
D	(2,8)
G	(8,9)
M	*tombstone*
Q	(2,4)

delete old SSTables

Outline: Cassandra Storage Engine

Writing Data: Buffering and Logging

Storing Data: SSTables

Reading Data

Compacting Data

- merge sort
- performance considerations
- HBase vs. Cassadra

Write Overheads

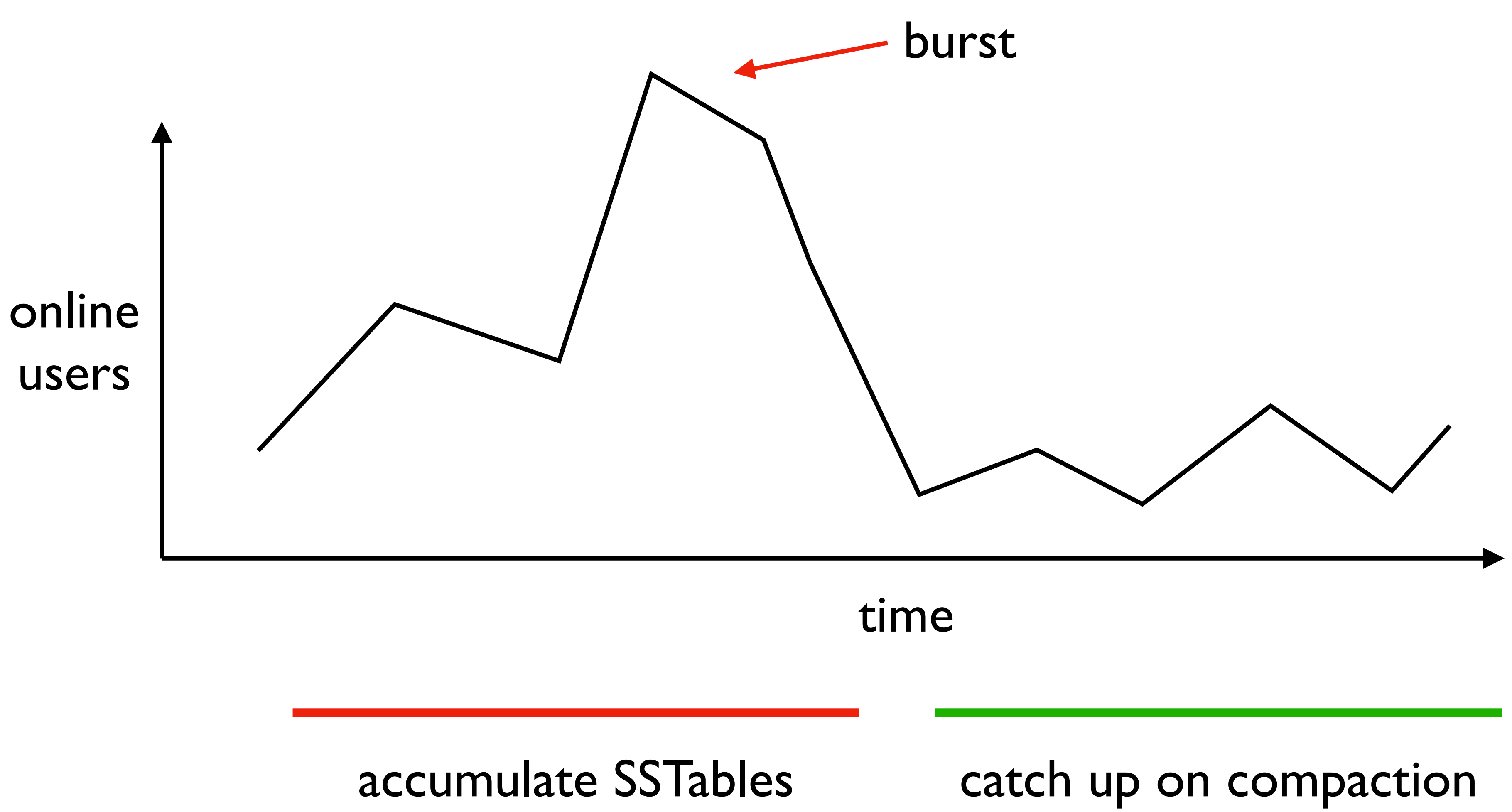
Compactions make reads faster but create write overheads.

Most written data gets re-written many times after the initial write (this is called "write amplification").

For example, in Facebook messages on HBase: "*Compaction causes about 17x more writes than flushing does, indicating that a typical piece of data is relocated 17 times.*"
~ Analysis of HDFS Under HBase: A Facebook Messages Case Study, Harter *et al.*

Background vs. Foreground Work

Compaction is background work, making LSM-based storage systems ideal for "bursty" workloads:



Compaction Policies

SizeTieredCompactionStrategy

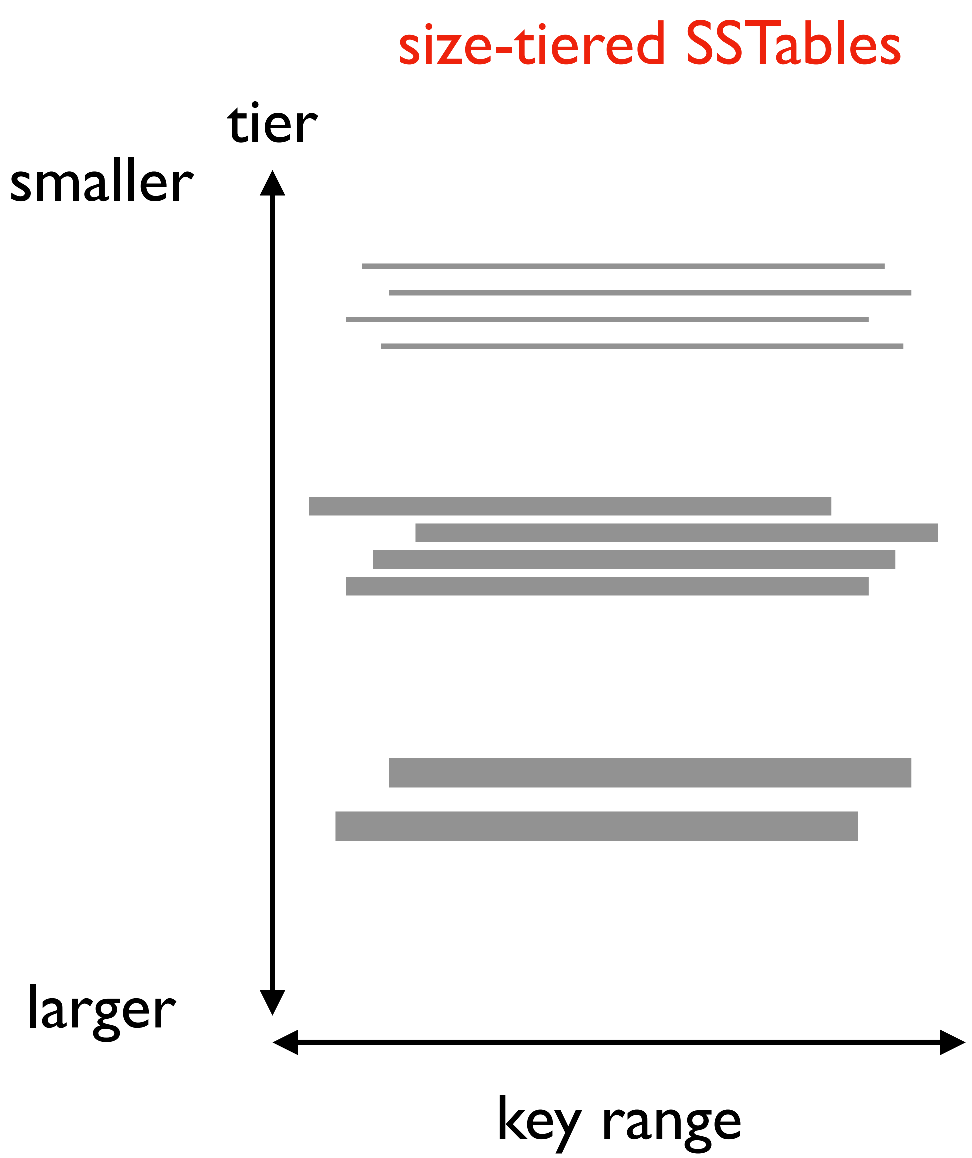
- optimized for write-heavy workloads
- try to compact SSTables of similar size together
- DEFAULT

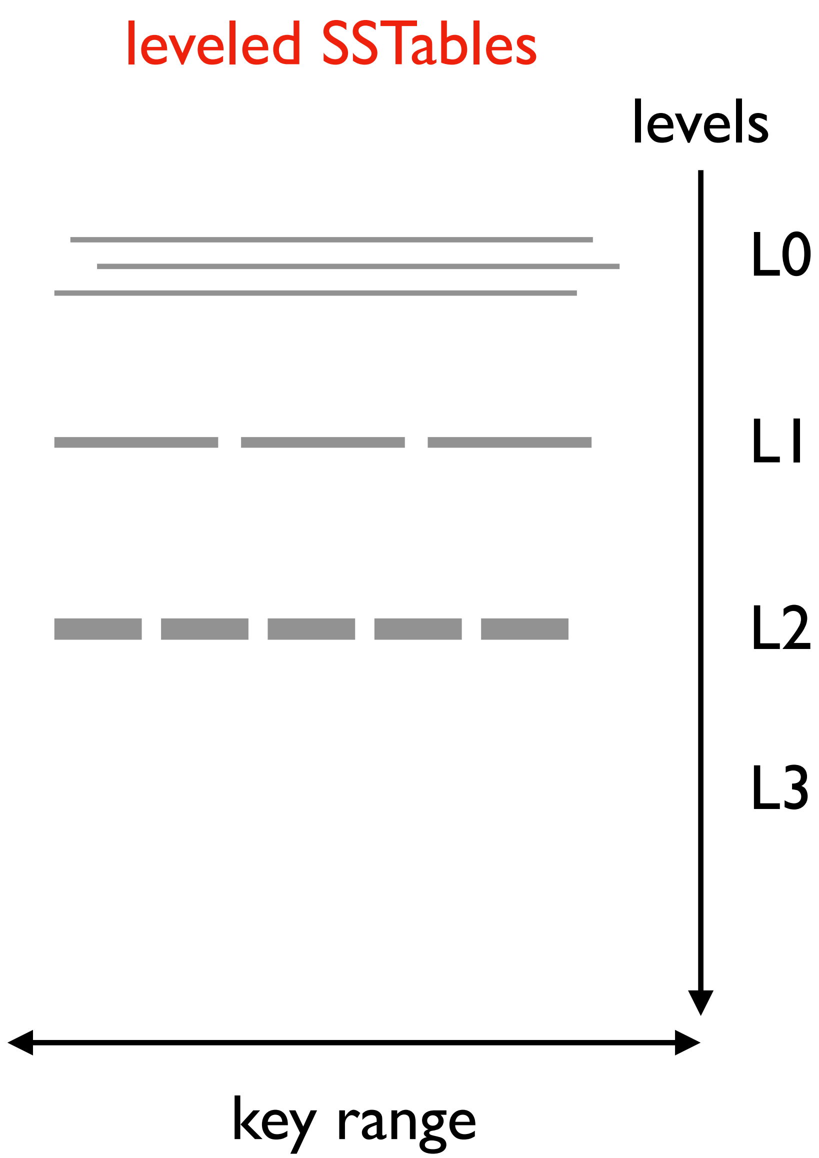
LeveledCompactionStrategy

- optimized for read-heavy workloads
- SSTables are assigned levels
- A key can appear in at most SSTable per level (except level 0)

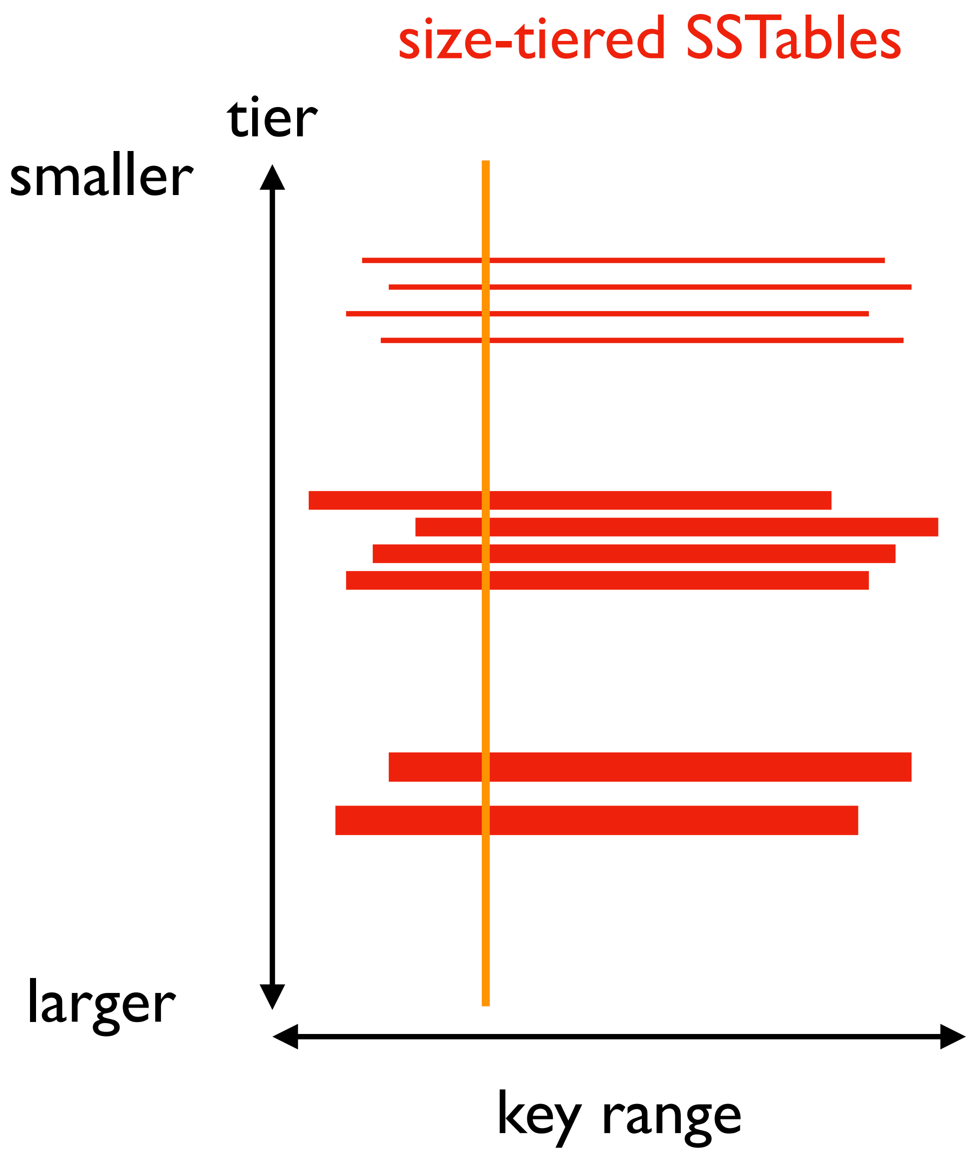
There are other strategies not covered in 544...

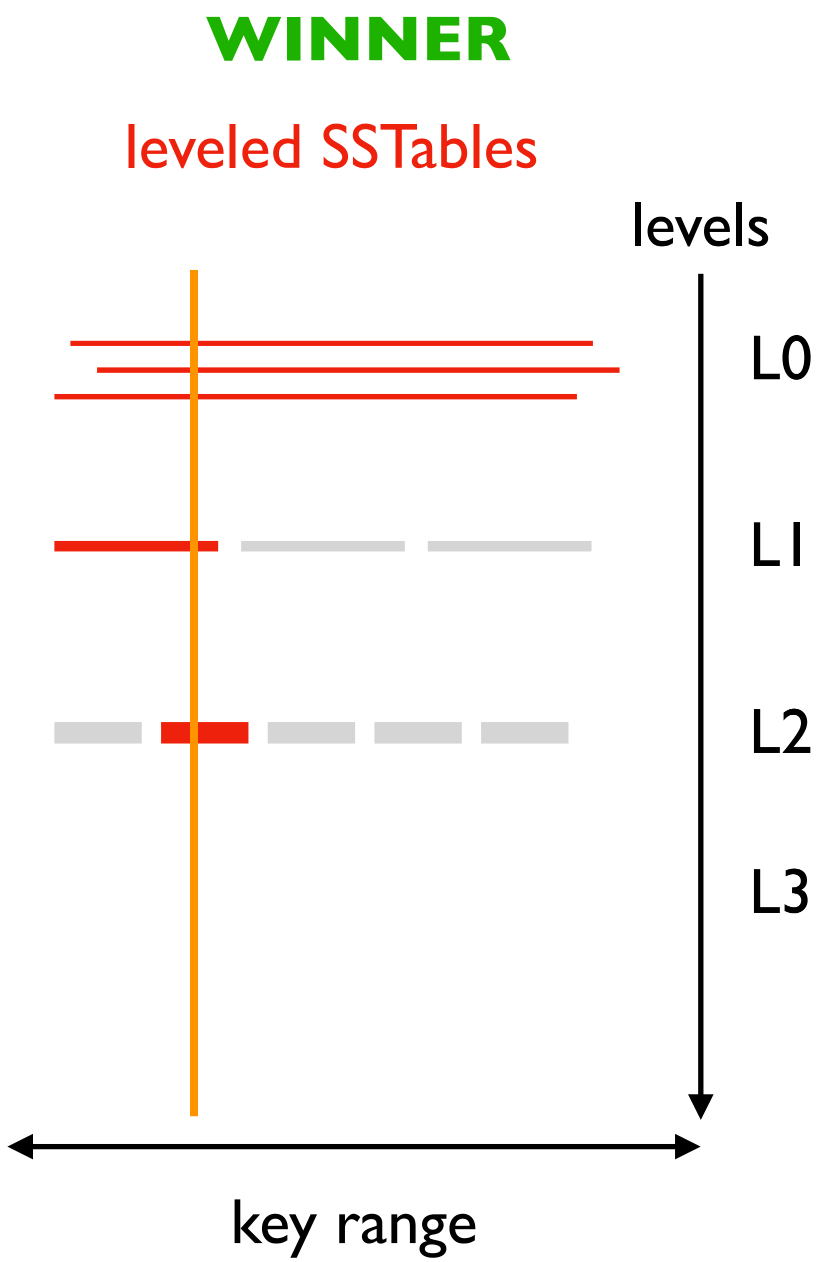
Compaction Policies





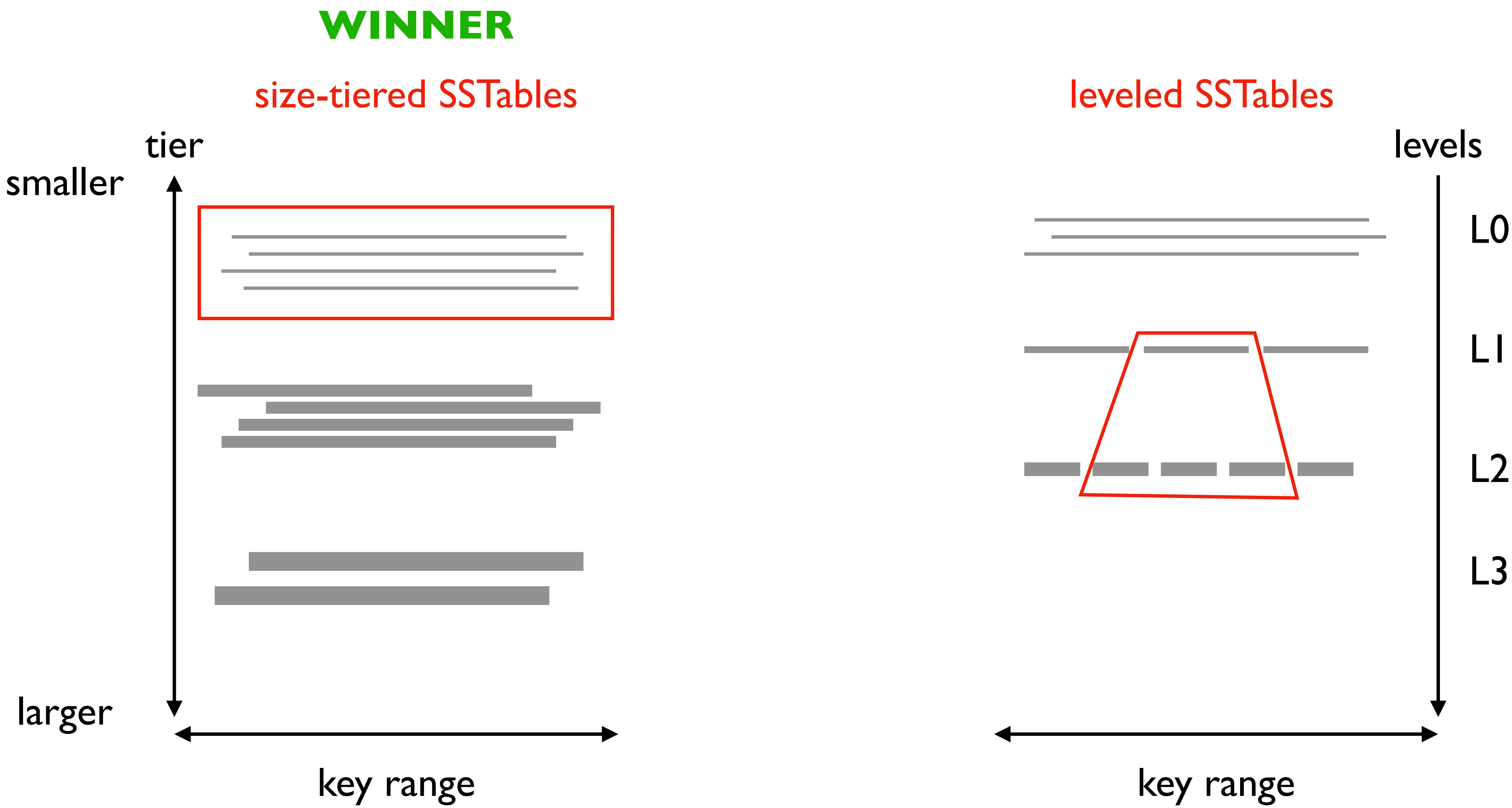
Reading a Key





leveled approach can avoid most SSTables during read

Compacting SSTables



size-tiered can compact similarly sized SSTables together
(more efficient than compacting very small files with very large files)

Outline: Cassandra Storage Engine

Writing Data: Buffering and Logging

Storing Data: SSTables

Reading Data

Compacting Data

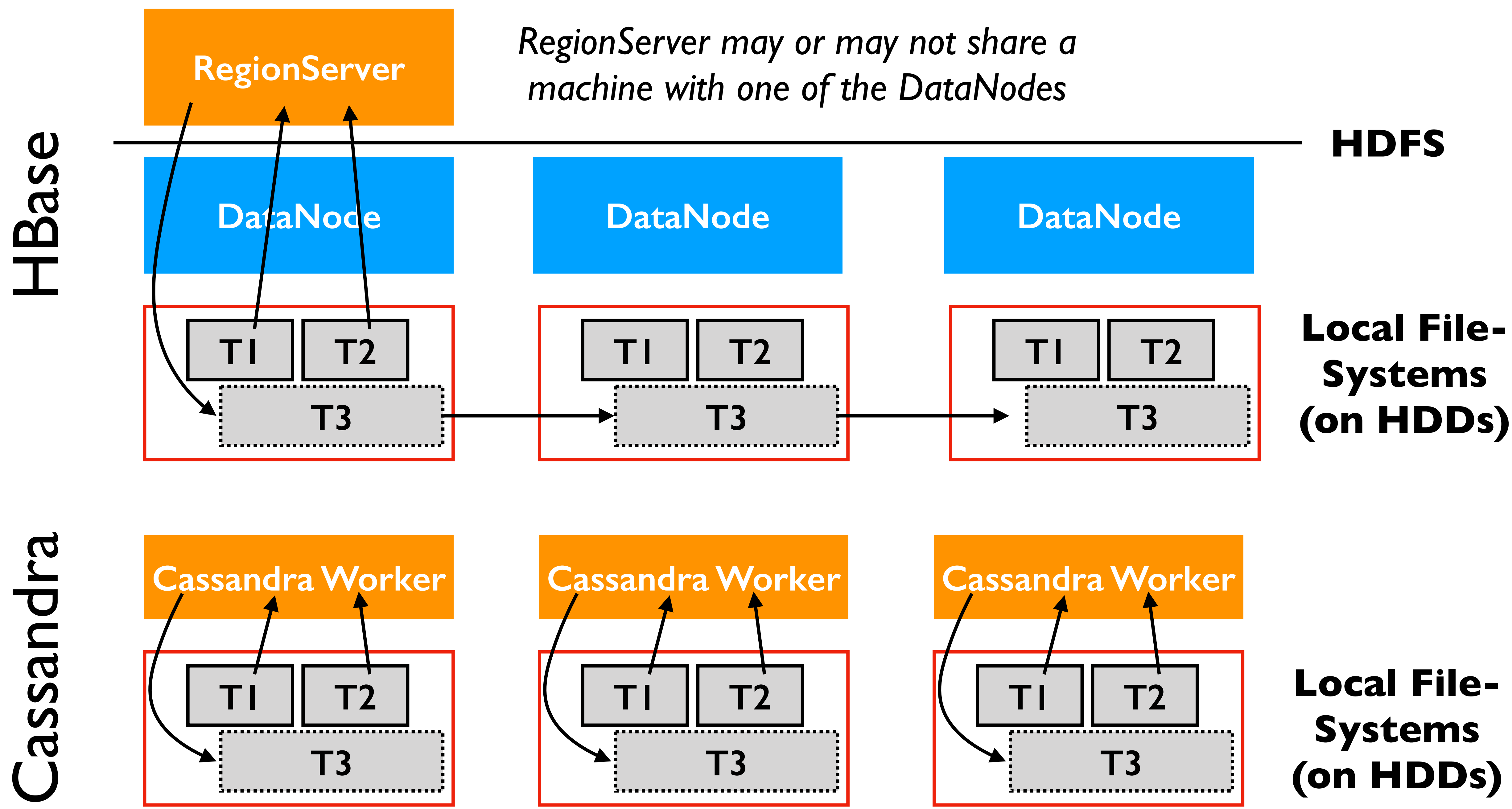
- merge sort
- performance considerations
- HBase vs. Cassadra

HBase vs. Cassandra

HBase	**Cassandra**
LSM	Replication
Replication	LSM

Which uses more resources for compaction?

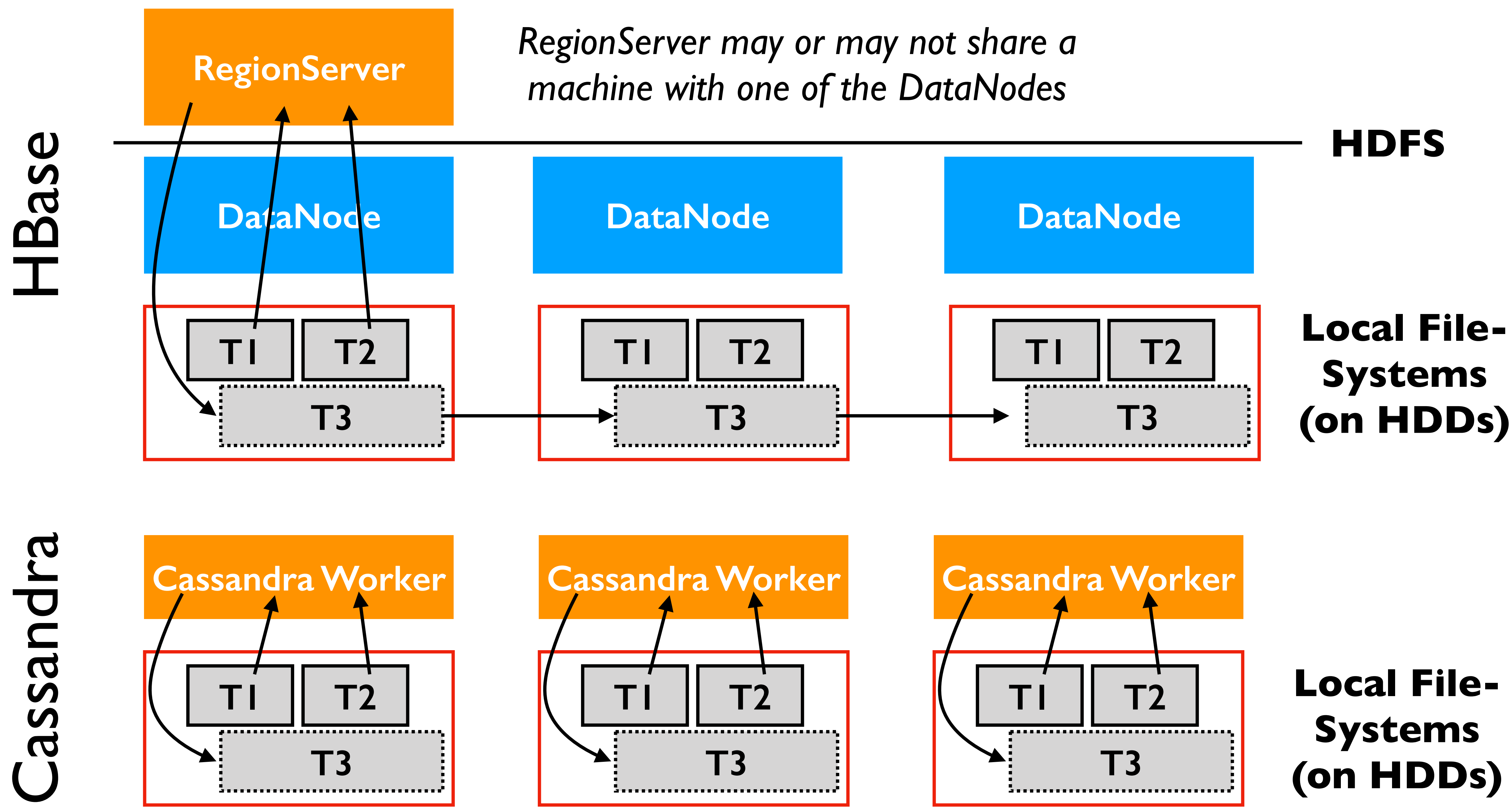
Scenario: compact T1 and T2 (both 1 MB) to produce T3 (2 MB)



note: this shows all 3 Cassandra LSMs in the same state, but they don't need to be in sync.

Which uses more resources for compaction?

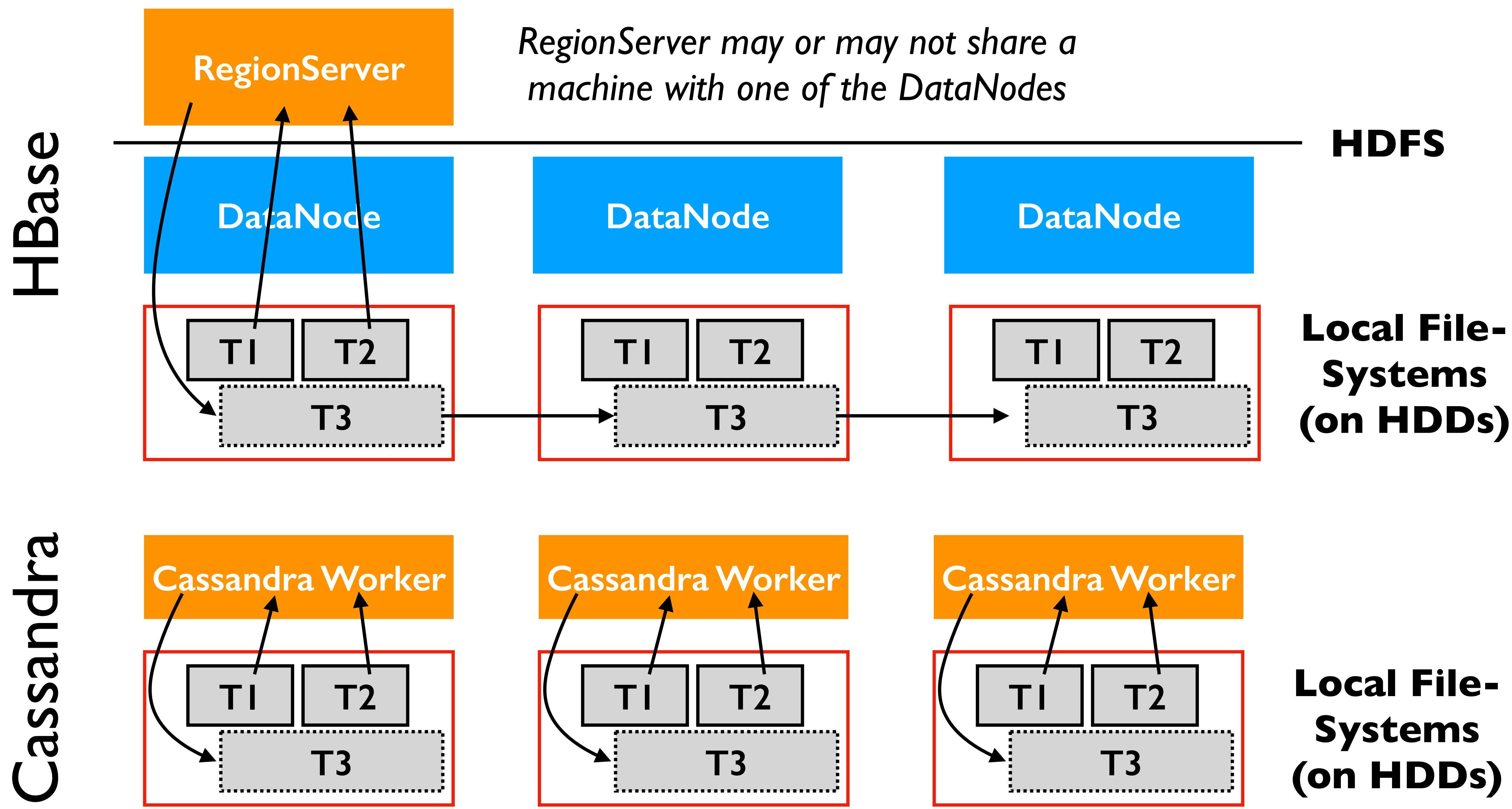
Scenario: compact T1 and T2 (both 1 MB) to produce T3 (2 MB)



compute: HBase merges 1x, Cassandra merges 3x. Winner: HBase

Which uses more resources for compaction?

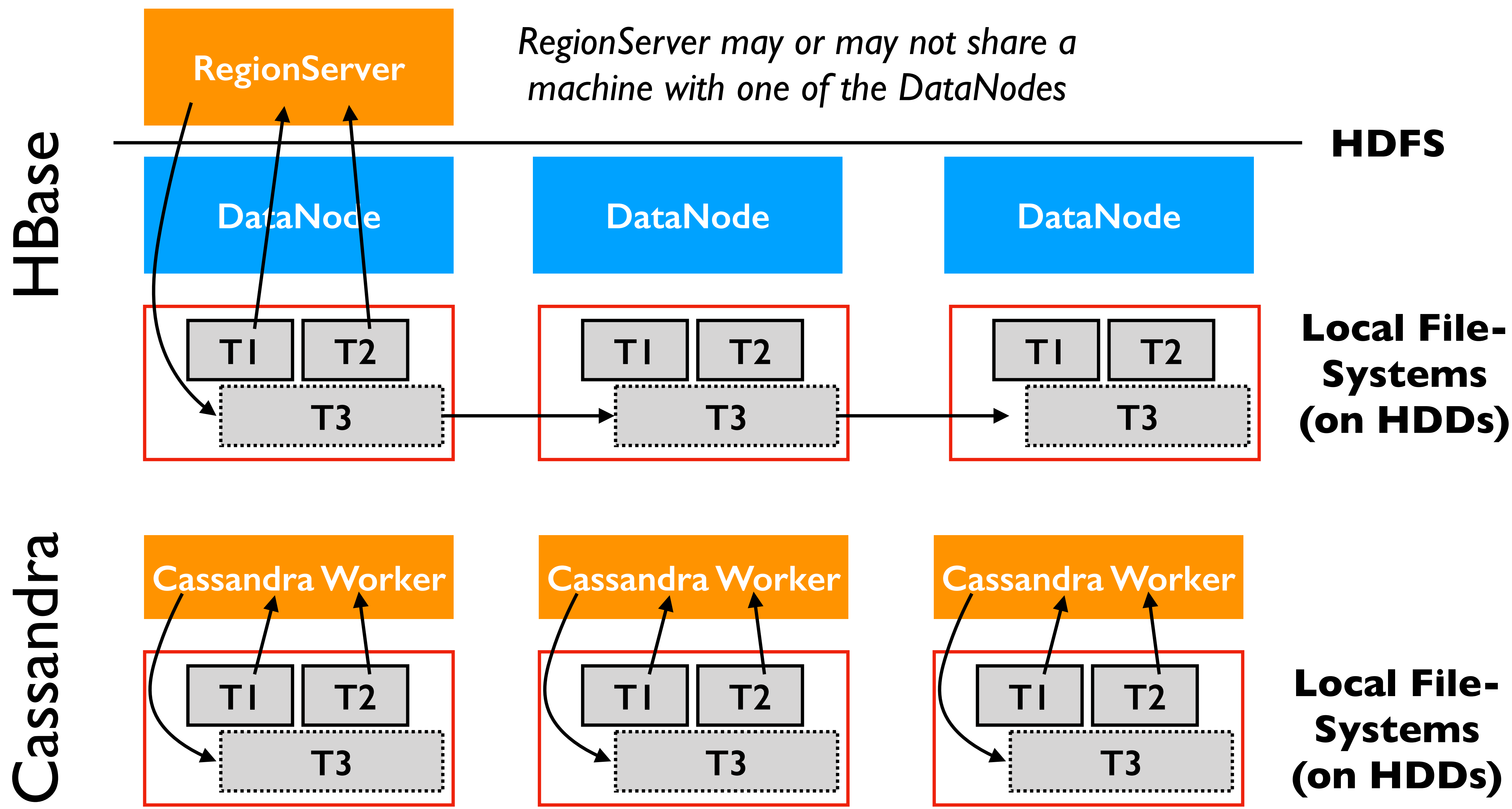
Scenario: compact T1 and T2 (both 1 MB) to produce T3 (2 MB)



disk reads: HBase uses ???? MB, Cassandra uses ???? MB. Winner: ????

Which uses more resources for compaction?

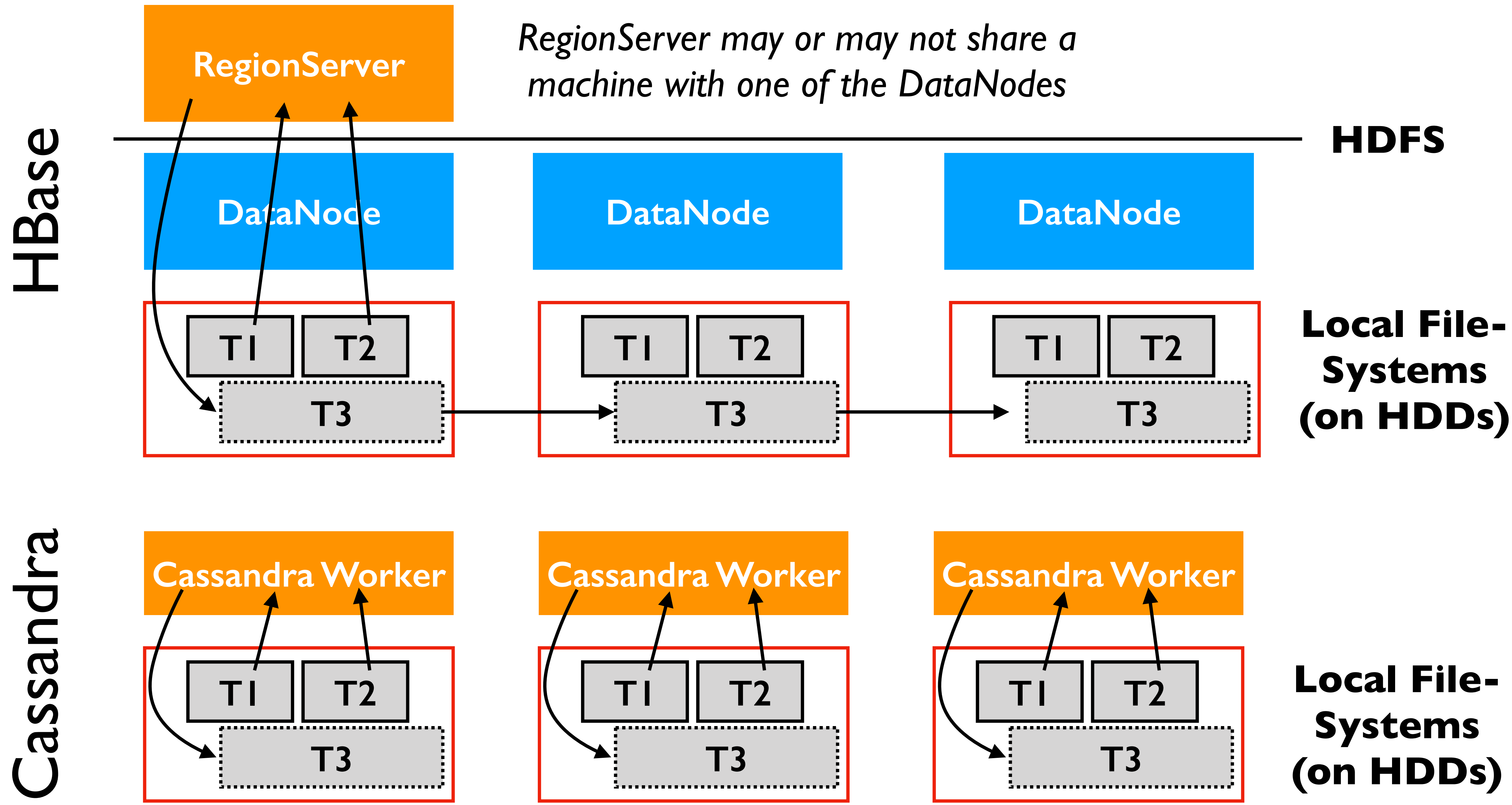
Scenario: compact T1 and T2 (both 1 MB) to produce T3 (2 MB)



disk reads: HBase uses 2 MB, Cassandra uses 6 MB. Winner: HBase

Which uses more resources for compaction?

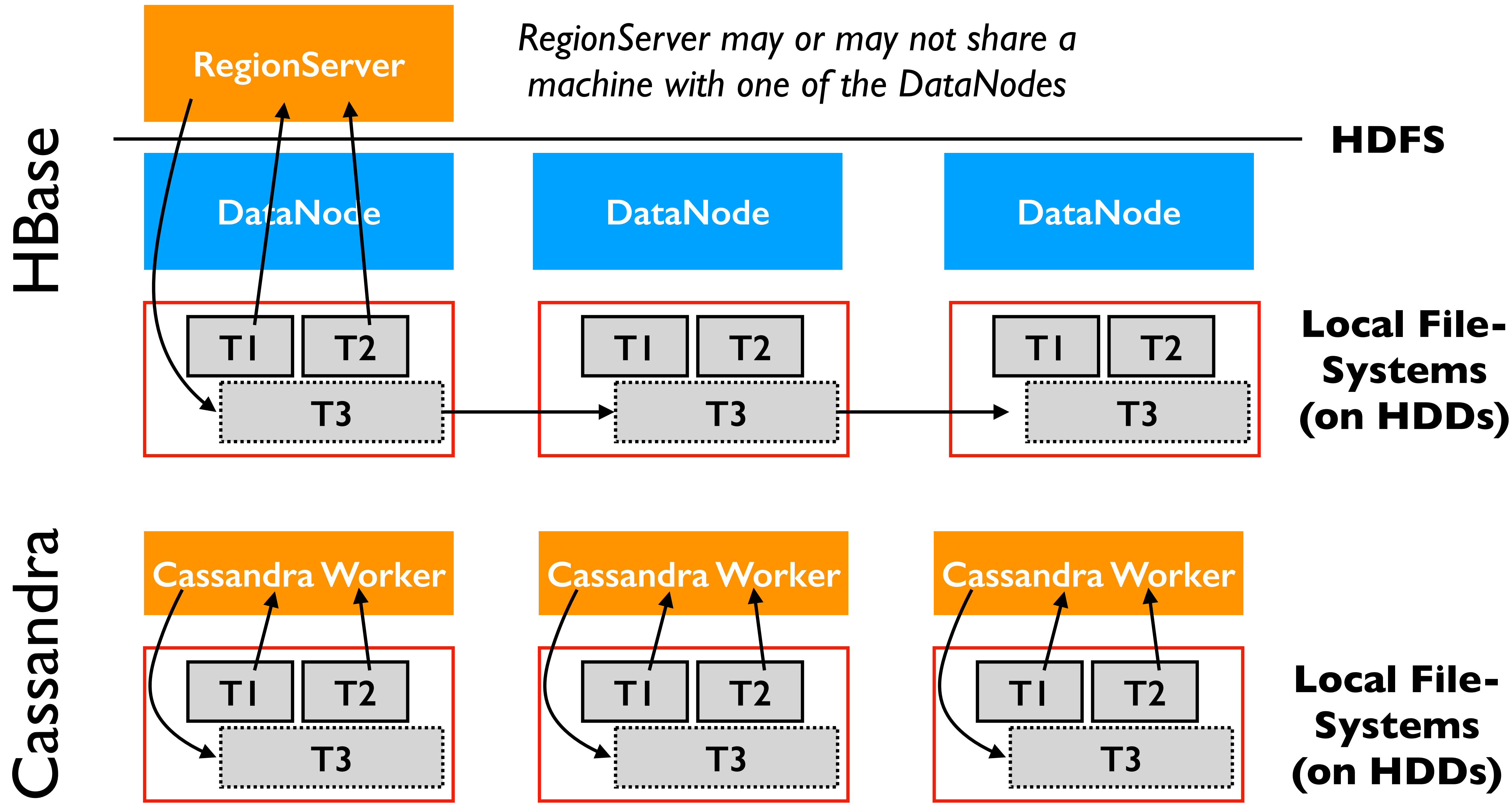
Scenario: compact T1 and T2 (both 1 MB) to produce T3 (2 MB)



disk writes: HBase uses ???? MB, Cassandra uses ???? MB. Winner: ????

Which uses more resources for compaction?

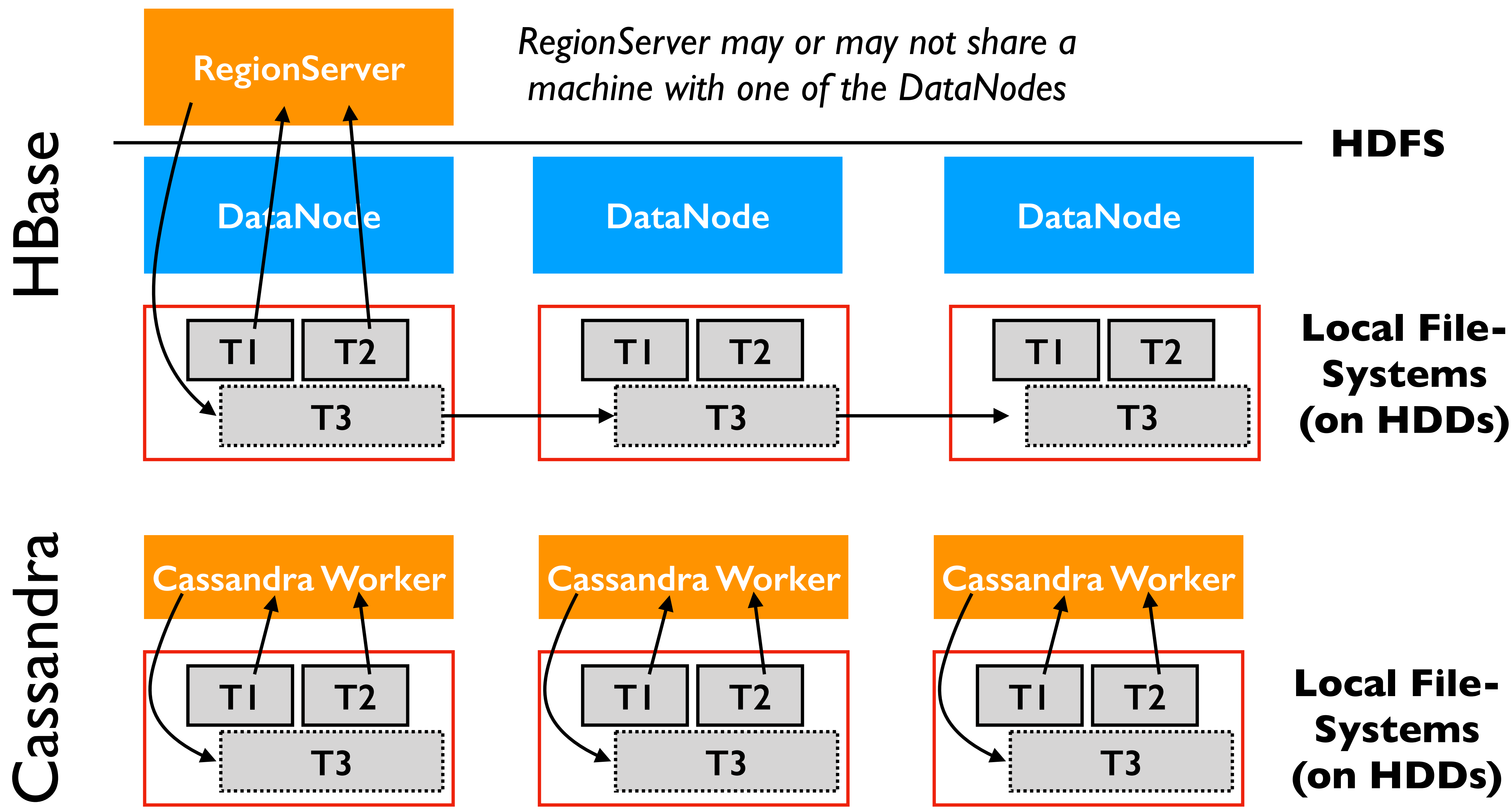
Scenario: compact T1 and T2 (both 1 MB) to produce T3 (2 MB)



disk writes: HBase uses 6 MB, Cassandra uses 6 MB. Winner: Tie

Which uses more resources for compaction?

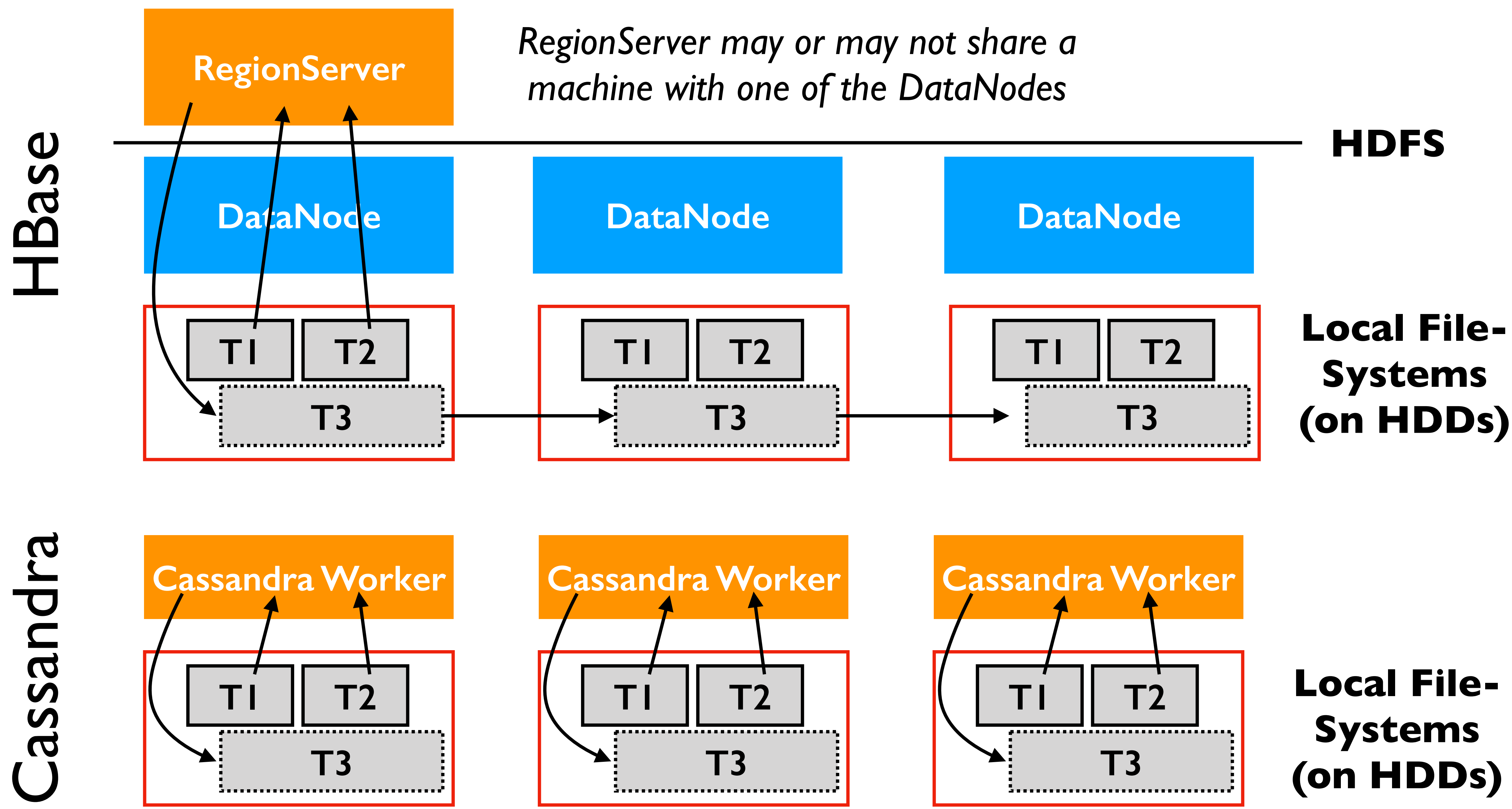
Scenario: compact T1 and T2 (both 1 MB) to produce T3 (2 MB)



network I/O: HBase uses ???? MB, Cassandra uses ???? MB. Winner: ????

Which uses more resources for compaction?

Scenario: compact T1 and T2 (both 1 MB) to produce T3 (2 MB)



network I/O: HBase uses 4-8 MB, Cassandra uses 0 MB. Winner: Cassandra

Architecture: HBase vs. Cassandra

HBase	**Cassandra**
LSM	Replication
Replication	LSM

HBase

- disk read efficient
- compute efficient

Cassandra

- network efficient
- flexible (different nodes can compact differently)