
Inside a concurrent index

Key takeaways

1. Leaf locks let independent point operations overlap. A scan that releases each lock after copying can combine rows from different moments.
2. A split changes where keys belong. Publish the new router before unlocking so a waiting writer can detect its stale route and retry.
3. Unlink empty leaves, then reclaim retired objects once readers are done. Deleting keys alone does not release the index's memory.

Common misunderstandings of concurrent indexes

Which claim would you defend right now? Choose before the evidence.

“A range scan that returns values the tree never held at the same moment is incorrect.”

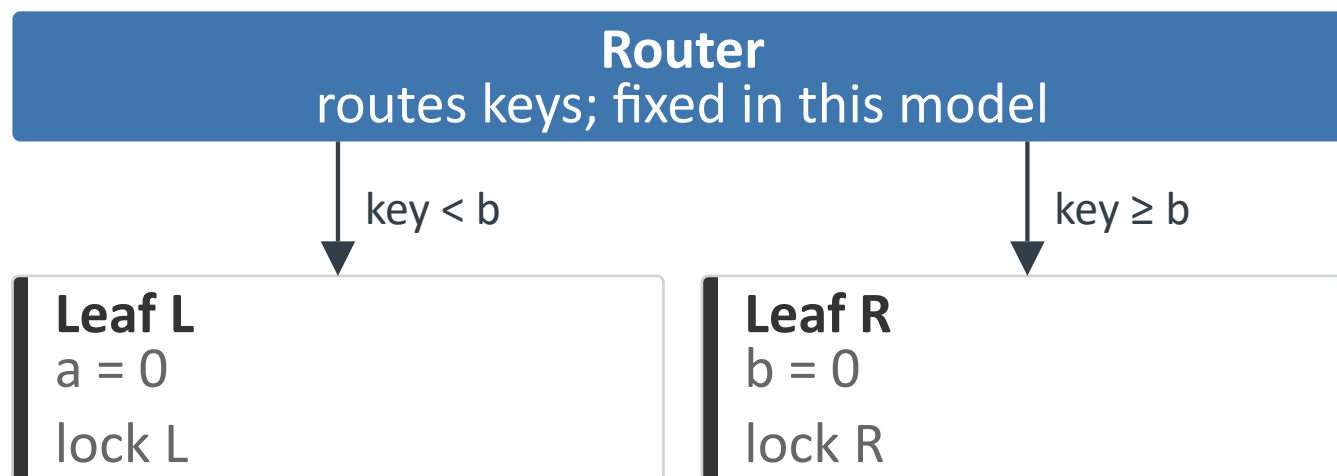
“Splitting a leaf is safe once the leaf’s lock is held.”

“Deleting every key from a leaf frees its memory.”

Keep your choice. Look for the mechanism or counterexample that would make it fail.

Two threads share one tree

Illustrative model: two fixed leaves, one writer, one scanner.



One process. Both threads use this same tree in memory.

Writer: put(a,1) locks L only; put(b,1) locks R only. Writers on different leaves never wait for each other.

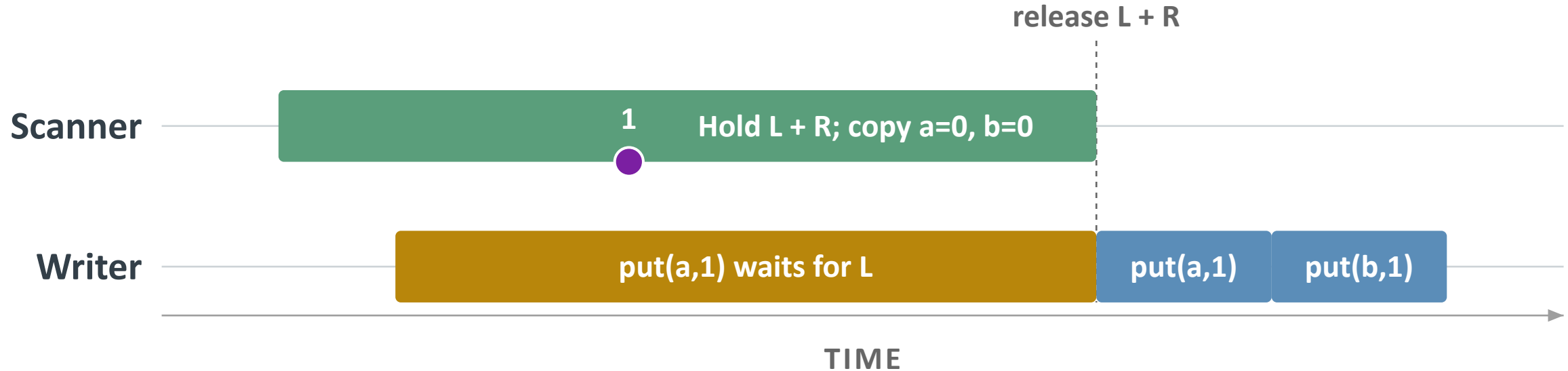
Scanner: visits L, then R. Copies rows into its own result list.

Each leaf has one lock. A thread must hold it to read or write that leaf.

Holding both locks gives a linearizable scan

Same two-leaf tree, initially $a=b=0$.

Acquire L, then R. Read both values before releasing either lock.

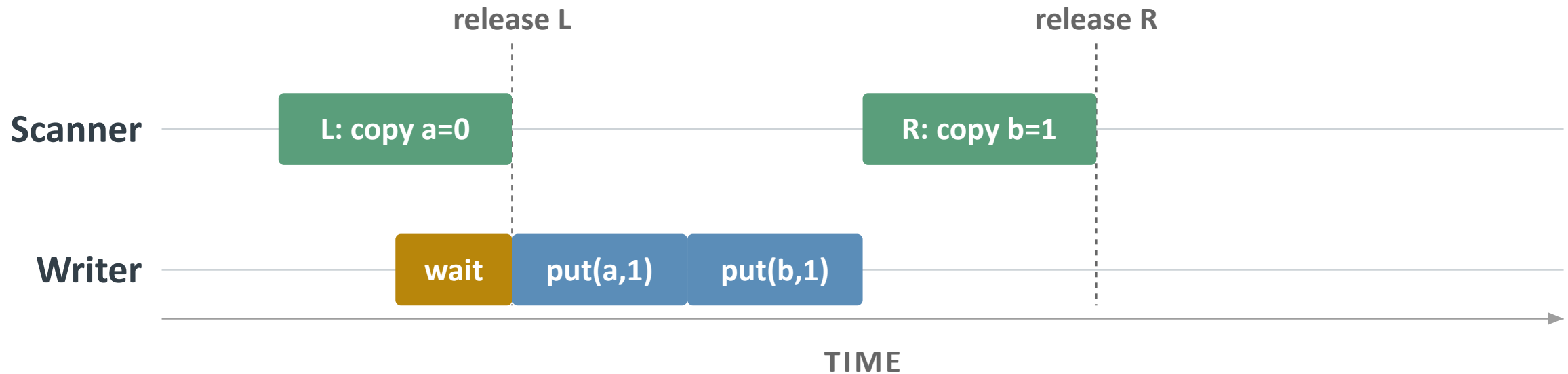


The purple dot is a linearization point. Both values stay fixed while the scan holds both locks.

Holding both locks serializes the scan with every writer to L or R.

Short leaf locks allow more concurrency

An alternative: release each leaf lock after copying that leaf's rows.



The writer resumes when L is released. Both writes finish before the scan returns.

Useful for a live status listing. A consistent total needs a snapshot.

More concurrency can improve performance by reducing writer stalls.

One lock at a time can mix tree states

Initially $a=0$ and $b=0$; one writer changes a to 1, then b to 1

Step	Thread and action	Shared tree (a,b)	Scanner's private rows
Start	Both values are zero	(0,0)	[]
1	Scanner: lock L, copy a, unlock L	(0,0)	[(a,0)]
2	Writer: lock L, set $a=1$, unlock L	(1,0)	[(a,0)]
3	Writer: lock R, set $b=1$, unlock R	(1,1)	[(a,0)]
4	Scanner: lock R, copy b, unlock R	(1,1)	[(a,0), (b,1)]

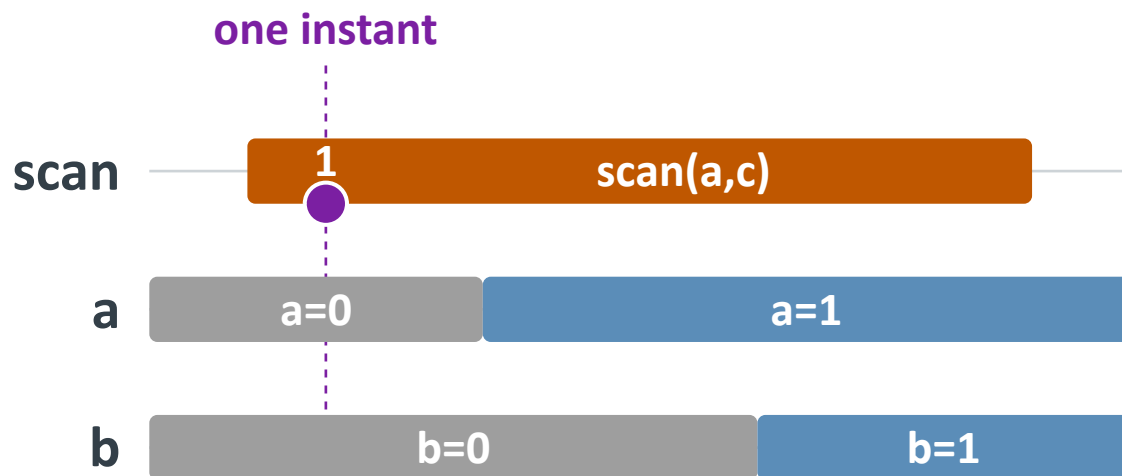
A linearizable scan must match one shared-tree state between its call and return.

The tree never holds $a=0, b=1$, so no linearization point can explain it.

The homework permits that mixed scan result

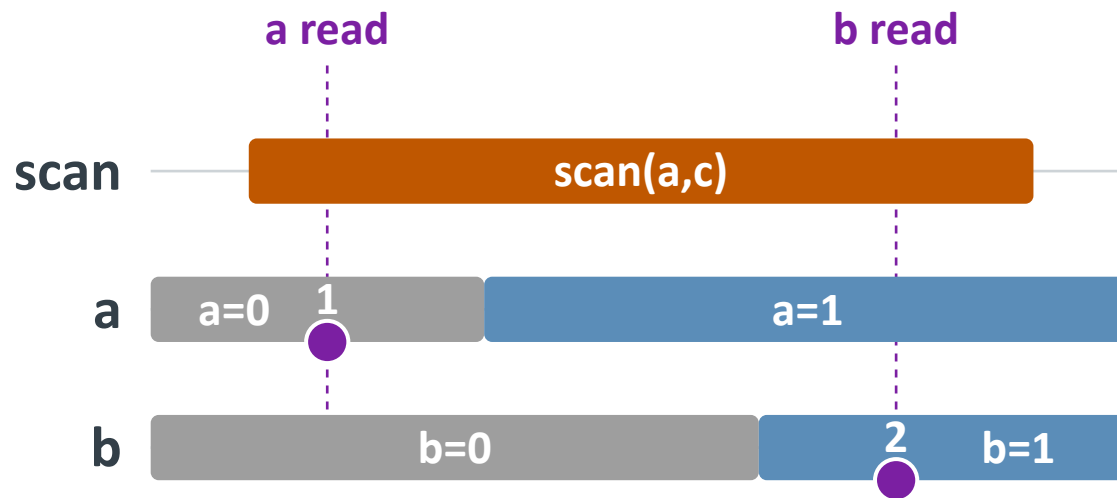
Point calls stay linearizable. A scan's rows need not share one instant.

Linearizable scan



$[(a,0), (b,0)]$

concurrent-index scan

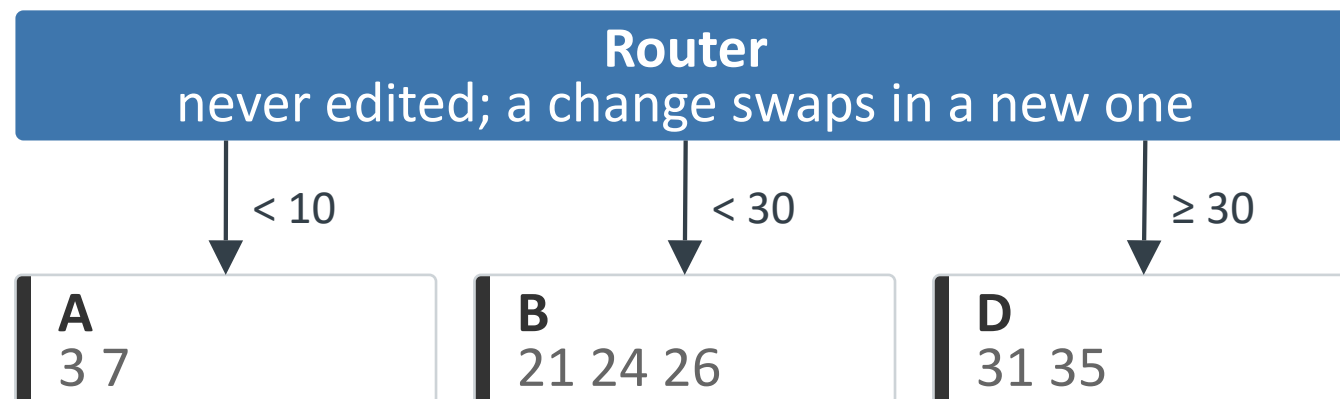


$[(a,0), (b,1)]$

The homework allows this: each returned pair existed at some instant during the scan.

The router changes: replaced, never edited

The two-leaf model fixed its router; splits and unlinks change routes. No thread edits a published router: a change builds a new one and swaps the pointer, so readers route with no lock.



get(24): load the router with no lock, route to B, lock B, then validate the route.

scan(5, 25): A then B, each under its own lock.

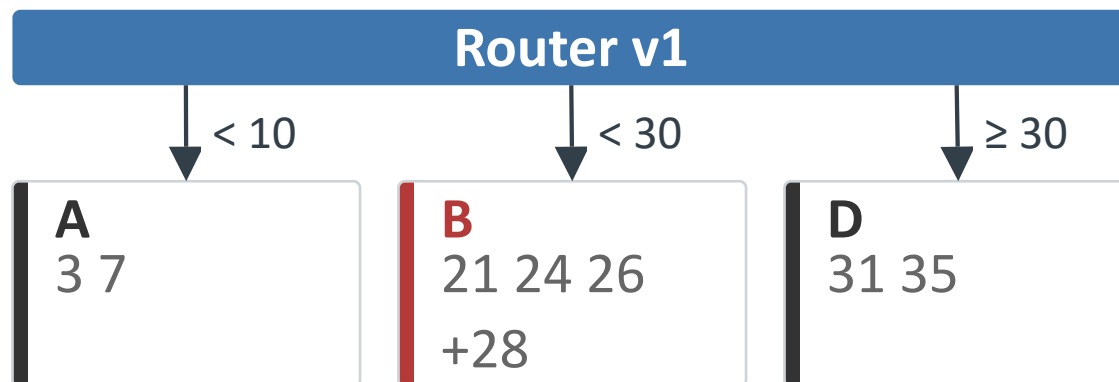
If a split moved the range, unlock and retry at the new leaf, as in Contention and progress.

Leaf locks still cover values. Changing routes means publishing a new router.

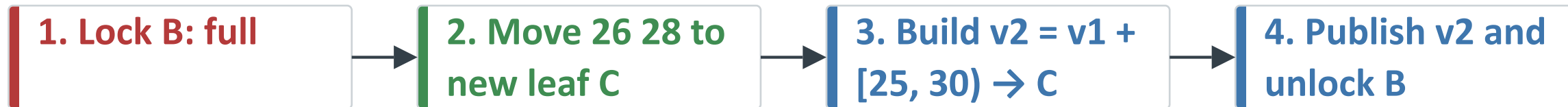
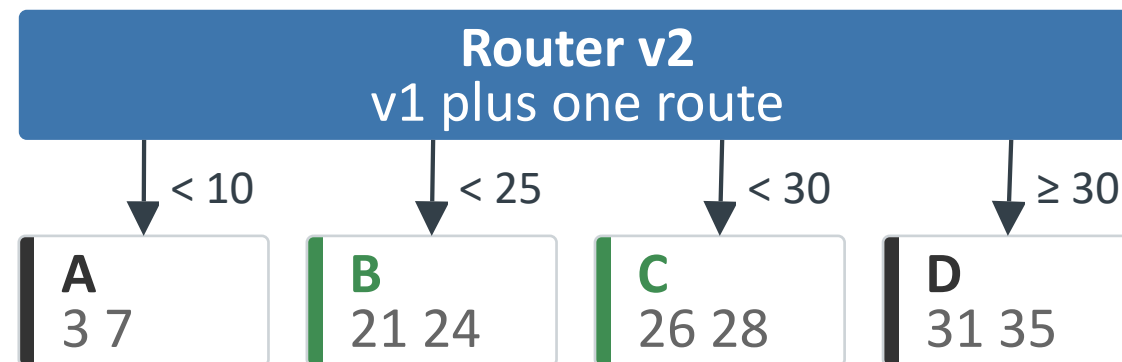
A split makes a new leaf and a new router

Leaves hold three keys; put(28) overflows B. The splitter holds B's lock throughout.

Before: put(28) overflows B



After: new leaf C, new router v2



Instapoll

Instapoll · Review · Oct 13 A

An index finds each key's leaf through one shared router. A writer locks its leaf, then re-reads the router to confirm the router still sends its key to that leaf, and retries if not. A full leaf splits: the thread holding the leaf's lock moves the upper half of its keys into a new leaf and builds a copy of the router that sends those keys there. In which order must the splitter finish so that every key stays findable?

- A. Unlock the leaf, then swap in the new router; the new leaf already holds its keys
- B. Swap in the new router, then unlock the leaf; a waiting writer re-reads the new one
- C. Either order; the writer's check under the lock catches every stale route
- D. Neither order; every writer must also lock the router while it inserts

Publish the split before releasing the leaf

Holding a leaf's lock does not prove that the key still belongs there.

put(27) waits for B. For key 27, the old router v1 says B; the new v2 says C.

Order	Unlock, then publish	Publish, then unlock
1. Splitter	Unlock B; v1 is current.	Publish v2; unlock B.
2. put(27)	Lock B; check v1. Insert 27 into B.	Lock B; check v2. Unlock B; retry and insert in C.
3. Splitter	Publish v2.	Already published.
4. get(27)	Search C: 27 is missing.	Search C: 27 is found.

Publish before unlocking so the waiting writer's route check sees the split.

Memory can grow with a fixed live-key count

Illustrative example: insert larger keys and delete the smallest keys at the same rate. Empty leaves stay in the router.

Time	A	B	C	D	E	Live keys	Allocated leaves
Start	1 2	3 4	5 6	—	—	6	3
Later	empty	empty	5 6	7 8	9 10	6	5

— means not allocated yet. Even after old readers finish, the current router still keeps A and B allocated.

Deleting a leaf's last key does not remove the leaf from the router.

Unlinking and reclaiming are separate steps

A replacement router skips empty leaves. Old readers may still hold pointers to those leaves.

State	Current router names	Memory for A and B
Empty, still linked	A B C D E	Needed by the current router.
Unlinked	C D E	Retired: old readers may still use it.
Reclaimed	C D E	Freed: no reader can still use it.

The new router assigns A and B's key ranges to C. C's existing keys stay put.

Publish before unlocking, as for a split. Retire the replaced router too.

Unlink from the current router. Reclaim only when no reader can still use the object.

Key Takeaways

1. Choose the scan contract before choosing lock duration. Holding both locks while copying the two fixed leaves gives one consistent state. Copying one leaf at a time lets writers proceed sooner but can mix states.
2. Check a route under the leaf lock and retry if it changed. Publish a split's new router before unlocking, so that check can detect the change. A router snapshot does not freeze the leaf values.
3. A fixed live-key count does not bound allocated leaves. Unlink empty leaves by replacing the router, then reclaim removed leaves and old routers after readers can no longer use them.

Diagnose a concurrent index

10 min

- Work in pairs. Produce four matches and one scan history; be ready to defend both.

1 MATCH: USE EACH REPAIR ONCE

Failure	Repair
A. Split unlocks before publishing.	1. Unlink empty leaves.
B. Router freed during a read.	2. Hold both scan locks.
C. Empty leaves stay linked.	3. Publish before unlocking.
D. Audit scan mixes tree states.	4. Reclaim after readers finish.

Matches: A → ____ B → ____ C → ____ D → ____

2 CONSTRUCT A SCAN HISTORY

Fixed leaves L: $a=0$ and R: $b=0$. Each access releases its leaf lock.

Writer ↓	Scanner ↓
put(a,1)	Copy a under L's lock
put(b,1)	Copy b under R's lock

- Order the accesses to return ($a=0$, $b=1$). Is that linearizable?
- Make the audit linearizable. Which writes must wait?