
concurrent-index: build a concurrent ordered index

Key takeaways

1. The contract is the assignment. Point operations are linearizable, scans follow stated rules while writes overlap, and memory safety counts as correctness.
2. Replacing the single lock is the performance work, but a wrong result under concurrency forfeits performance credit, so the correctness argument comes first.
3. Three captures: V1 fixes the design and its correctness, V2 adds delete and measured prototypes, and V3 keeps one index in service with bounded memory.

A one-sentence ask hides a systems assignment

“Make this sorted map concurrent and fast” is easy to say

Simple request

Replace a locked BTreeMap with something that scales across threads. It sounds like a weekend of lock tuning.

Never treat the one-line ask as the spec.

Real artifact

One ordered map over byte keys, shared by every thread, that owns its synchronization and serves range scans.

Name the protocol and its invariants first.

Hidden difficulty

Every legal interleaving must return a legal result, and stay that way as the contents move.

Budget your time here, not on lock tuning.

An index that is fast but sometimes returns an impossible answer is wrong.

SPEC.md, section by section

Read four yourself; “What grading builds” is a checklist for your agent

Section	What it answers	How to use it
API and correctness	What a right answer is, and what a wrong one costs	Read it yourself, beside the trait’s doc comments. Your tests come from here
What you may build on	What counts as an index you built	Read before you choose a design
What grading builds	Which files are graded and which dependencies resolve	Reference. Have your agent check it before it adds a file or a dependency

SPEC.md, section by section

Read four yourself; “What grading builds” is a checklist for your agent

Section	What it answers	How to use it
Workloads and grading	What is measured, and on which capture	Read the capture table; use the workload list to question your design
Operating conditions	The sizes and the memory rule your design must survive	Read before DESIGN-v1.md, even though memory is scored at V3

The spec is short because the Rust doc comments in `crates/index-api` carry the exact contract. Read both; neither is complete alone.

Words the spec assumes you know

Each has a plain meaning; this week's lessons teach the ideas behind them

Term	Plain meaning
capture	The one commit per stage that the course grades
point operation	get, insert, or update: one key per call. A scan covers a range of keys
sealed case	A test the course runs and never shows you
workload family	A kind of graded work, such as scan-heavy; each earns its own performance credit
baseline	The supplied single-lock B-tree that every performance score is compared with

Words the spec assumes you know

Each has a plain meaning; this week's lessons teach the ideas behind them

Term	Plain meaning
throughput, p99 latency	Operations finished per second; the time 99% of operations finish within
scaling	How throughput changes as worker threads are added
pinned physical cores	Real, full-speed cores the grader binds its threads to; no hyperthreads
live keys	The keys in the index now; overwritten and deleted data is not live
requested live bytes	Bytes your code has allocated and not yet freed, as the allocator counts them

Five methods, all on &self

The harness shares one index across every thread; there is no outside lock

crates/index-api — frozen; its doc comments are the contract

```
pub trait Index: Send + Sync {
    fn register_thread(&self) -> ThreadGuard<'_>;
    fn get(&self, key: &[u8]) -> Option<Vec<u8>>;
    fn insert(&self, key: &[u8], val: &[u8]);           // upsert
    fn update(&self, key: &[u8], val: &[u8]) -> bool;  // only if present
    fn scan(&self, lo: &[u8], hi: &[u8],              // [lo, hi)
            sink: &mut dyn FnMut(&[u8], &[u8]) -> bool);
}
```

- Each thread registers before its first call; dropping the guard marks it quiescent
- update never creates a key: absent means false and nothing inserted
- scan visits keys in bitwise order and stops as soon as the sink returns false
- The key and value slices a sink receives are borrowed for that call only
- Raw-pointer fields need unsafe impl Send/Sync; review the agent's SAFETY argument

Correct means one sequential story, with a looser rule for scans

The linearizability lecture's contract, applied to this API

Operation	What the contract requires
get, insert, update	Linearizable: each call appears to take effect at one instant between its call and its return
scan, no overlapping write	Exactly the keys in $[lo, hi)$, ascending
scan, overlapping writes	Ascending and duplicate-free; every returned pair existed at some time during the scan; every in-range key present for the whole scan appears, unless the sink stopped early
every operation	Memory safety is part of correctness, not a separate quality

A scan need not be one instant, so it may mix rows from different moments. It may not skip a key that was there the whole time.

The starter is correct; it just holds one lock

Every writer excludes every reader, so threads mostly wait

The supplied starter

```
pub struct StudentIndex {  
    map: RwLock<BTreeMap<  
        Vec<u8>, Vec<u8>>>,  
}  
// insert: map.write()  
// scan:   map.read() for the  
//         whole range
```

What the spec rules in and out

allowed:

- std collections as parts
- sync, allocation, reclamation
- building blocks (allowlist)
- a coherent hybrid you built

forbidden:

- RocksDB, LMDB, SQLite, ...
- a crate that IS the index
- picking an index per benchmark
- the starter or a bare BTreeMap

Performance is measured against a paired single-lock baseline. Beating it means letting operations that do not conflict run at the same time, and then proving that the ones that do conflict still tell one story.

The allowlist is building blocks, not an index

What each crate in the spec's table is for, in the table's order

Crate	What it gives you
arc-swap	An atomically swappable Arc; a reader loads a snapshot
atomic	Atomic<T> for a plain type with no padding bytes
crossbeam	Re-exports the three below, plus channels
crossbeam-epoch	Epoch-based reclamation: a pinned guard delays frees
crossbeam-queue	Concurrent queues, bounded and unbounded
crossbeam-utils	CachePadded for hot fields; scoped threads
once_cell	One-time initialization and lazy values
parking_lot	Mutex and RwLock that never poison; a Mutex is one byte
rand	Random keys and schedules for your tests

The SPEC tells the agent where judgment lives

Each rule closes a shortcut an agent would otherwise take

Spec line	Why it matters
all keys coexist in one index; scans give one order	a hybrid may route by key, but it is still one index a scan can cross
no detecting the benchmark	the grade measures the design, not a recognizer for the workload
only index-student's Cargo.toml and src/ are copied	other crates, build scripts, and rust-toolchain.toml do not exist at grading
an exact-version dependency allowlist	grading builds offline; an unadmitted dependency means nothing builds
no #[global_allocator]	the course supplies the allocator and measures memory through it
DESIGN-v1.md before V1 code	the protocol and its invariants are written down while they can still be wrong

Two races your design must answer

DESIGN-v1.md asks for both; work out the legal outcomes before writing the code

update vs. insert on an absent key

```
T1: update(k, "u") -> ?
T2: insert(k, "i")
then: get(k) -> ?
```

```
legal:   (false, "i")  update first
          (true,  "u")  insert first
illegal: (true,  "i")
          (false, "u")
```

scan vs. ascending inserts

```
preload: a, c, e
T1: scan(a, z)
T2: insert b, then d
```

```
must appear: a, c, e
may appear:  both, either,
              or neither of b, d
never:       out of order, a key
              twice, a key never
              inserted
```

Returning d without b is legal: the scan can pass b's position before b exists. Each table is an oracle your stress test can check without knowing the interleaving.

Validation starts before implementation

For each claim, ask what evidence would change your mind

Claim	Evidence
point operations are linearizable	racing threads record calls and results; check that one legal order explains them
scans follow the overlap rules	preloaded keys all appear; output strictly ascending; every new row was really written
no lost or phantom update	absent-key races only ever produce the two legal outcomes
no data race or aliasing bug	tiny-threshold concurrent tests under cargo +nightly miri test
the index scales	repeated paired runs against the baseline across thread counts
scans do not starve writers	writer and scanner rates reported separately, never one total

The eight supplied tests are single-threaded. Concurrent tests are yours; the graded checker is sealed.

Correctness gates performance

Speed only counts where the answers are right

- Correctness and performance carry most of the grade; the writeup and lesson slides carry the rest
- A workload family earns no performance credit if the final capture gets it wrong or does not finish
- A result no legal ordering allows, anywhere, forfeits all performance credit
- Performance is measured on the final capture against the single-lock baseline; your local curves do not predict it
- Reading the supplied comparators
 - A ratio is the median of per-pair ratios, not a ratio of medians
 - The pair comparator runs a fixed amount of work; the interference tool counts what finishes in a fixed time window

A slower design you can argue is correct is worth more than a faster one you cannot.

Plan for the operating conditions, not one benchmark

The graded mixes, sizes, and seeds are sealed; the conditions are not

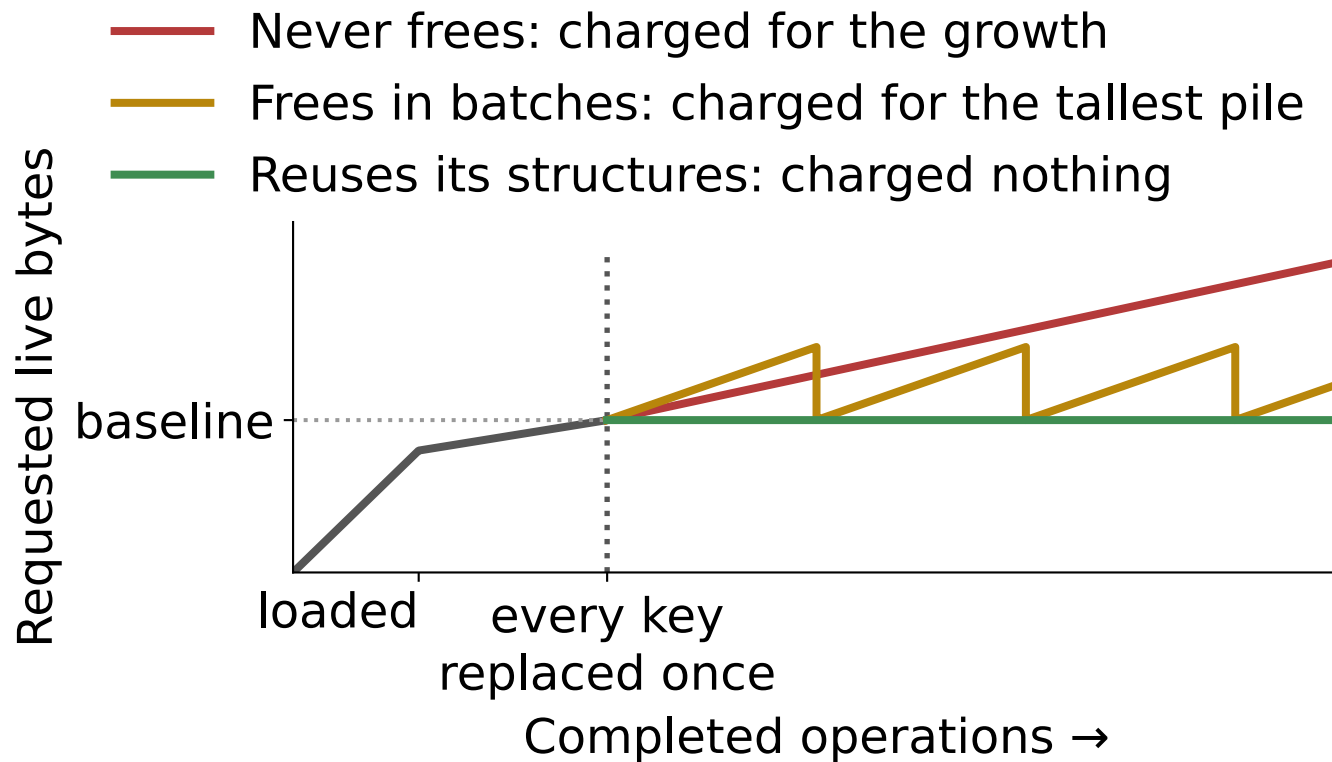
- Point-heavy, mixed, and scan-heavy work, over fixed and growing populations
- Access is never skewed toward a fixed set of keys; growing workloads may favor recent inserts, wherever they land
- Long scans run while other threads keep writing, and both sides' service counts
- Different key forms and lengths can share one live index

Live keys	Key	Value	Payload	Scan rows
≤ 2,097,152	≤ 1,024 B	≤ 128 B	≤ 256 MiB	≤ 32,768

Ask of your design: where do new keys land, what does a long scan hold while writers wait, and what do you store per key when a key is a kilobyte?

Once warm, memory should stay level

Sustained service is scored on the final capture; SPEC.md states the rule now



The baseline is taken after the load and one replacement of every live key. Structures set up by then are not charged.

From there, growth and the peak above the baseline are each charged beyond a small allowance.

A sustained history completes more than 1,000 operations per live key, so anything kept per operation adds up.

Bounded is not enough: storage you will free later is charged while you hold it.

Often you do not know what is broken

A concurrency bug usually shows up as a rare failure

- Symptom
 - A stress test fails once in two hundred runs; the agent reruns it and it passes.
- Bad first response
 - Call it flaky, add a sleep or a retry, and move on.
- Better first response
 - Record the failing history: each call, its result, its start and end.
 - Ask which invariant in DESIGN-v1.md that history violates.
 - Shrink it to the fewest threads and keys that still fail, and keep it as a test.

One passing run does not rule out the losing schedule; one failing run proves it exists.

Three captures, and what each one retires

Capture	What arrives or is judged	The assumption it retires
V1	DESIGN-v1.md before code, then V1 correctness	(baseline: keys arrive and are overwritten, never removed)
V2	Single-key delete, measured prototypes, a frozen DESIGN-v2.md	“nothing ever leaves the index”
V3	Sustained service: one index, a long history, live keys moving through new ranges, bounded memory	“a short run shows steady state” and “freed later is free”

Each later spec reaches your repository only after you pin the stage before it. The V1 spec already names what is coming, so a V1 design that cannot grow a delete or bound its memory is a V1 design problem.

What to commit, and when

Use the deadlines Canvas shows

- Commit DESIGN-v1.md before changing index code, with 4–6 acceptance tests
- Each design file locks at its capture: DESIGN-v1.md at V1, DESIGN-v2.md (a template until V2) at V2. A later change of design goes in WRITEUP.md; editing a locked file holds your grade for instructor review
- Tag agent-initial when the agent's first StudentIndex passes cargo test, before you read its code

What to commit, and when

Use the deadlines Canvas shows

- Edit only crates/index-student/; grading copies its Cargo.toml and src/, nothing else
- Commit at task boundaries: the writeup cites commits, and one squashed commit leaves nothing to cite
- Push V1 to main, submit that SHA to the V1 checkpoint, then git pull for the V2 materials

From the repository root

```
cargo build
cargo test
cargo run --release -p index-student --example compare
```

Key Takeaways

1. The contract is the assignment. Point operations must be linearizable, a scan under writes may mix moments but may not skip a key present throughout, and memory safety is correctness. Write the legal outcomes down before the code.
2. Correctness gates performance. The single lock is correct and slow; your design lets non-conflicting work overlap, and a wrong answer under concurrency costs the performance credit that overlap was meant to earn.
3. Design V1 for all the operating conditions and all three captures. Delete arrives in V2 and bounded memory over a long history in V3, and the V1 spec already tells you both are coming.