Build a search engine for your inbox

Key takeaways

1. The index is the assignment — design the local SQLite layout; let the agent type the SQL.
2. update must equal rebuild, and search must never write.
3. Two releases: Build, then Resilience breaks the assumptions Build allowed.

A one-sentence ask hides a systems assignment

A search engine for mail is easy to ask for and hard to make correct

Simple request

Index an mbox corpus and search for messages by token. One sentence, and it sounds like a weekend.

Never treat the one-line ask as the spec.

Real artifact

A SQLite-backed CLI with a schema, update logic, tests, and behavior that survives a crash.

Name the schema and the CLI contract first.

Hidden difficulty

Mailbox mutation, idempotent updates, read/write boundaries, crash recovery, and performance.

Budget your time here, not on parsing.

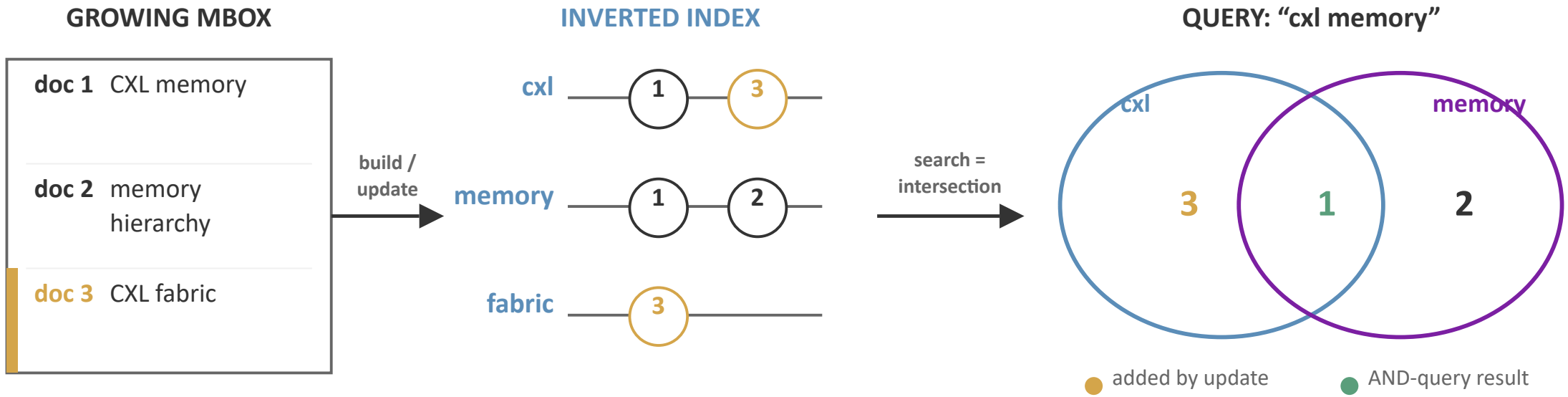
The assignment is not parsing email. The assignment is making state stay correct as the world changes.

Search is the easy part; the index is the work

mindex keeps a fast search index in sync with a mailbox that keeps growing

- RFC/MIME defines message fields; the assignment fixes parsing, tokenization, query behavior, and graded identity
- The implementation chooses the local SQLite layout and update bookkeeping; real mail also needs a fallback when Message-ID is missing or duplicated

A query is one set intersection; build and update must keep every posting correct.



Everything it touches is an mbox

An mbox is just messages concatenated, each behind a From-separator line — the real format Thunderbird and Gmail Takeout write.

One record in mail/inbox.mbox — then a blank line, the body, the next From_

```
From alice@example.edu Thu Jan 15 09:00:00 2026
From: alice@example.edu
Subject: CXL memory pooling
Message-ID: <m1@example.edu>
```

The CLI you implement (build / update / search / show / info)

```
mindex build  --db idx.db mail/          # index from scratch
mindex update --db idx.db mail/          # pick up newly appended mail
mindex search --db idx.db "cxl memory" --after 2026-01-01
```

You vs. the Agent: email-index

Your Responsibility
The local SQLite layout — tables, indexes, and keys
The update design: incremental == rebuild
Stable docids + a real-mail fallback when Message-ID fails
Acceptance tests + the DESIGN.md decisions

Agent's Responsibility
The SQL: tables, inserts, the AND query
The CLI glue (argparse, --json output)
Wiring in the provided parser.py
Boilerplate, refactors, test scaffolding

Shared: parsing is PROVIDED — reimplementing harness/parser.py earns nothing

The SPEC tells the agent where judgment lives

Specification is not over-control

Spec line	Why it matters
search must be a pure read	prevents hidden update work under concurrent search
update must be idempotent	running it twice cannot duplicate indexed mail
FTS5 is forbidden	students build the inverted index rather than wrap SQLite’s built-in full-text engine
crash recovery is graded	a process death is a test case, not an excuse
DESIGN.md explains the choice	the human owns the schema and the update strategy

Build your own inverted index — FTS5 is forbidden

FTS5 is SQLite's built-in full-text-search engine — one CREATE VIRTUAL TABLE and it would build this whole index for you. That is exactly why it is banned: the inverted index — token → the messages that contain it — IS the artifact. An AND query is a set intersection.

Local SQLite layout — the assignment leaves this open

```
CREATE TABLE messages(
  docid      INTEGER PRIMARY KEY,
  message_id TEXT, sender TEXT,
  day        TEXT, subject TEXT);

CREATE TABLE postings(
  token TEXT, docid INTEGER);
```

AND query = intersect the postings

```
SELECT docid FROM postings
 WHERE token = 'cxl'
INTERSECT
SELECT docid FROM postings
 WHERE token = 'memory';
```

update must equal rebuild — and search never writes

- Rebuild-equivalence: after appends + update, search == a fresh build
- Idempotent: running update twice equals running it once
- search never writes — bundling an update into every read is the classic bug

The tempting mistake

```
def search(q):
    update()         # a write
    return query(q)  # inside a read!
# fails the read-purity check
```

Read and write are separate

```
def search(q):       # pure read
    return query(q)

def update(corpus):  # explicit, idempotent
    for m in appended_since(mark):
        index(m)
```

Validation starts before implementation

For each claim, ask what evidence would change your mind

Claim	Evidence
update only touches the delta	row-count diff plus changed-folder log
search is a pure read	no write transaction during search; clean mtime before/after
reconciliation handles deletion	message removed from mbox disappears from postings
crash recovery is safe	kill at injected points; recovered DB matches oracle
performance is acceptable	benchmark on staged corpus, not one tiny sample

Two releases: Resilience breaks Build's assumptions

Release	New reality	The Build shortcut it breaks
Build	The mailbox only grows — new mail is appended	(baseline — the shortcut you're allowed)
Resilience	Mail gets deleted and moved in place	"update only ever appends"
Resilience	A crash can interrupt any write	"my update always runs to completion"
Resilience	Updates race other updates and searches	"my update is the only writer"

Resilience is one sealed drop — its spec and checks stay hidden until the Build checkpoint closes. That is how real specs evolve: a new reality breaks an assumption you were allowed to make.

Often you do not know what is broken

The first symptom is usually just wrong-shaped behavior

- Symptom
  - Only 15 new emails arrived, but update takes a long time.
- Bad first response
  - Ask the agent to make update faster without naming what changed.
- Better first response
  - Ask what work update believes it is doing.
  - Measure folder changes, row counts, writes, and lock time.
  - Compare one surprising case to a known-small delta.

Debug the model of the work before optimizing the code.

The random challenge

Test your agent’s code with one specific example

Pick one
One real message, one token, one date filter — your pick, not a case from the agent’s own test suite.

Know the answer first
Get ground truth by hand — grep the raw mbox. The expected answer must come from outside the code under test.

Compare, then generalize
A mismatch is never one bug: name the class of inputs it represents, then add the case to your acceptance tests.

The agent’s tests pass by construction — a hand-checked case tests reality

```
$ grep -l "Subject: CXL memory pooling" mail/*.mbox   # ground truth, by hand
mail/inbox.mbox
$ mindex search --db idx.db "pooling" --json          # the code under test
[]                                      <- one specific example found the bug
```

Optional capstone — index your real mail, read-only

- Ungraded and separately packaged — not a beat or a checkpoint
- mailfetch.py pulls a bounded slice of your inbox into a local mbox
- Read-only by construction — it cannot delete, move, or even mark mail read
- The random challenge, live: one weird real message breaks what no synthetic tier did

Read-only IMAP discipline (forbidden: STORE/APPEND/EXPUNGE/COPY/MOVE)

```
# EXAMINE = read-only open; never SELECT (read-write)
imap.examine("INBOX")
# BODY.PEEK: fetch the body without setting the Seen flag
typ, data = imap.uid("FETCH", uid, "(BODY.PEEK[])")
imap.logout()
```

source: the capstone bundle — CAPSTONE.md + mailfetch.py

Key Takeaways

1. The index is the assignment — design the local SQLite layout and update state; let the agent type the SQL
2. update must equal rebuild, and search must never write — invariants, not features
3. Resilience arrives sealed and breaks Build's assumptions; the optional capstone points mindex at your real mail