# SQL you actually write for mindex

Key takeaways

1. Write the query first; schema and indexes exist to serve it.
2. EXPLAIN QUERY PLAN is the evidence that your index is used.
3. One transaction per promise, or search loses a stored message.

# Common misunderstandings of SQL, indexes, and commits

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

"A column named in WHERE will be served by any index on that column."

"Committing after every statement is safer than batching work in one transaction."

"When COMMIT returns, the bytes are in the main .db file."

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

# One CLI command becomes one query

The command surface is fixed by the spec; the SQL behind it is yours

**What the user runs**

```
mindex search "cxl memory" \
  --after 2026-01-01 --json
```

**What your tool has to ask SQLite**

```
SELECT d.docid, d.subject
  FROM postings p
  JOIN docs d ON d.docid = p.docid
 WHERE p.token IN (?, ?)
   AND d.sent_day >= ?
 GROUP BY d.docid
HAVING COUNT(DISTINCT p.token) = 2
 ORDER BY d.sent_day DESC;
```

- Every flag in the spec is a clause: the sender filter is a WHERE, the date filters bound one column.
- An AND query is one row per token per message, regrouped back to one row per message.
- Design the schema by writing this query first, not by listing the fields a message has.

source: hw/email-index/SPEC.md: the CLI you must implement

# SQL: Structured Query Language

Describe the result you want; let SQLite choose the access plan

- SQL is a declarative language: state what data you want.
- SQLite chooses an access plan from the schema and available indexes.
- The same core syntax appears in PostgreSQL, MySQL, and other databases.

SQL is a small language for asking structured questions about structured data.

# The docs table: W2-Thu’s constraints put to work

Declared once in CREATE TABLE; enforced on every insert that follows

**A strict SQLite table**

```
CREATE TABLE docs (
  docid    INTEGER PRIMARY KEY,
  msg_id   TEXT UNIQUE,
  sender   TEXT NOT NULL,
  sent_day TEXT NOT NULL,
  subject  TEXT
) STRICT;

INSERT INTO docs(sender, sent_day)
VALUES ('ada@example.com', 20260101);
-- error: 20260101 is not TEXT
```

- docid INTEGER PRIMARY KEY: the row identity postings will reference.
- msg_id UNIQUE: re-indexing a message is a caught error, not a silent duplicate.
- sent_day NOT NULL: the date filter and the index order key can rely on it.
- STRICT: the mistyped insert fails at write time, not at query time.

source: https://sqlite.org/lang_createtable.html; https://sqlite.org/stricttables.html

# The postings table is the artifact

An inverted index is one row per (token, message), not a text column to scan

**token → the messages containing it**

```
CREATE TABLE postings (
  token TEXT NOT NULL,
  docid INTEGER NOT NULL
    REFERENCES docs(docid)
    ON DELETE CASCADE,
  PRIMARY KEY (token, docid)
) WITHOUT ROWID, STRICT;
```

postings

| token | docid |
|---|---|
| cxl | 41 |
| cxl | 77 |
| memory | 41 |
| memory | 93 |

- PRIMARY KEY (token, docid) is the index: every row for one token is stored adjacent.
- WITHOUT ROWID stores the row in that key order instead of duplicating it in a side index.
- FTS5 is forbidden because this table is the assignment; the schema and key choice are yours.

source: hw/email-index/SPEC.md: you build the inverted index yourself; https://sqlite.org/withoutrowid.html

# Four statements are the whole CRUD vocabulary

Use parameters for values; never build SQL by concatenating user input

**Create and read**

```
INSERT INTO docs(msg_id, sender, sent_day)
VALUES (?, ?, ?);

SELECT docid, subject
  FROM docs
 WHERE sender = ?
 ORDER BY sent_day DESC;
```

**Update and delete**

```
UPDATE docs
   SET subject = ?
 WHERE docid = ?;

DELETE FROM docs
 WHERE docid = ?;
```

- Question-mark placeholders keep values separate from SQL commands.
- WHERE chooses the affected rows; omit it and UPDATE or DELETE affects every row.
- Searching by sender is this SELECT with one more bound parameter, not a new code path.

# The insert path keeps index == corpus

Every posting you fail to write is a message search can never find again

**Indexing one parsed message**

```
cur.execute(
    "INSERT INTO docs(msg_id, sender, sent_day) VALUES (?, ?, ?)",
    (rec.message_id, rec.sender, rec.sent_day),
)
docid = cur.lastrowid
cur.executemany(
    "INSERT OR IGNORE INTO postings(token, docid) VALUES (?, ?)",
    [(token, docid) for token in set(rec.tokens)],
)
```

- set(rec.tokens) writes one posting per distinct token; keep counts only if you rank on them.
- executemany prepares the statement once and binds many rows, instead of N round trips.
- INSERT OR IGNORE makes re-indexing the same message harmless, which update depends on.

source: hw/email-index/SPEC.md: parsing is provided; the index is not

# Instapoll

Instapoll · Review · Sep 15 B

Your search filters by sender and orders by sent day, and EXPLAIN QUERY PLAN prints “SCAN docs”. What does that tell you?

- A. No index the planner can use covers this query’s filter
- B. The table is too small for SQLite to store an index on it
- C. The query returned its rows in the wrong order for the client
- D. The database needs a checkpoint before the plan can change

# An index avoids a full-table scan

Put equality filters first, then the column that supplies order

**The application query**

```
SELECT docid, subject
  FROM docs
 WHERE sender = ?
 ORDER BY sent_day;
```

**The matching access path**

```
CREATE INDEX docs_sender_day
    ON docs(sender, sent_day);

-- rows for one sender are adjacent
-- and already ordered by sent_day ASC
```


**Two single-column indexes can bring the sort back; the composite prefix does both jobs.**

source: https://sqlite.org/queryplanner.html#1_6_multi_column_indices

# EXPLAIN QUERY PLAN is the evidence

An index you created is not always the index the planner used

**Before the index**

```
EXPLAIN QUERY PLAN
SELECT docid, subject FROM docs
 WHERE sender = ? ORDER BY sent_day;
QUERY PLAN
|--SCAN docs
`--USE TEMP B-TREE FOR ORDER BY
```

**After CREATE INDEX docs_sender_day, same query**

```
QUERY PLAN
`--SEARCH docs USING INDEX
   docs_sender_day (sender=?)

-- no SCAN, and no sort afterward
```

| Plan line | What it means |
|---|---|
| SCAN docs | every row is read, whatever the filter says |
| SEARCH … USING INDEX | the index seeks to the matching rows |
| SEARCH … USING COVERING INDEX | the index alone answers the query; the table is never read |
| USE TEMP B-TREE FOR ORDER BY | the rows had to be sorted afterward |

source: https://sqlite.org/eqp.html

# One consistency promise, one transaction

A message and its search postings must become visible together

**One atomic update**

```
BEGIN;

INSERT INTO docs(...);
INSERT INTO postings(token, docid)
VALUES ('cxl', 41), ('memory', 41);

COMMIT;
```

**Two commits can tear the index**

```
INSERT INTO docs(...);
COMMIT;

-- crash here: message exists,
-- but search can never find it

INSERT INTO postings(...);
COMMIT;
```

The transaction boundary is the consistency boundary.

source: https://sqlite.org/lang_transaction.html; hw/email-index invariant: index == corpus

# WAL is why that COMMIT survives the crash

Enough to run mindex safely now; the internals are Week 5

**Set once, per database**

```
PRAGMA journal_mode = WAL;

-- one writer and many readers
-- proceed at the same time
```

| File | What it holds |
|---|---|
| mindex.db | the last checkpointed database |
| mindex.db-wal | committed pages not yet folded in |
| mindex.db-shm | the shared index into that log |

- COMMIT appends to the -wal file and syncs it; the main .db file is not current until a checkpoint.
- Copying only the .db file out from under a live writer therefore loses committed messages.
- The grader fingerprints every file matching your database path, so the sidecars are part of the artifact.

source: https://sqlite.org/wal.html; hw/email-index/SPEC.md: the harness fingerprints PATH*

# Key Takeaways

1. Write the query the spec asks for first; the schema and the index exist to serve it.
2. EXPLAIN QUERY PLAN is the only evidence that the index you created is the one being used.
3. One transaction per consistency promise; WAL makes that commit survive, and W5-Thu's WAL internals deck explains how.