# Structured Data & SQLite

Key takeaways

1. SQLite puts a transactional database behind a local file.
2. Tables, keys, and constraints make structure executable.
3. A committed change survives a crash; a half-written file does not.

source: adapted from cs.utexas.edu/~witchel/371M/lectures/sqlite-notebook-slides.pdf

# Common misunderstandings of SQLite

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

“If the data is in a file, a crash cannot leave it half updated.”

“A database always means running and administering a separate server.”

“A schema is only documentation; the database cannot enforce it.”

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

# Your app needs to remember things

- App data lives in memory: variables, lists, and objects.
- When the process dies, everything in memory is gone.
- Users expect their data to survive restarts.
- Persistent storage is a design choice: what kind fits the data?

**Key-value**
Small settings: theme, login token

**Files**
Unstructured blobs: images, documents

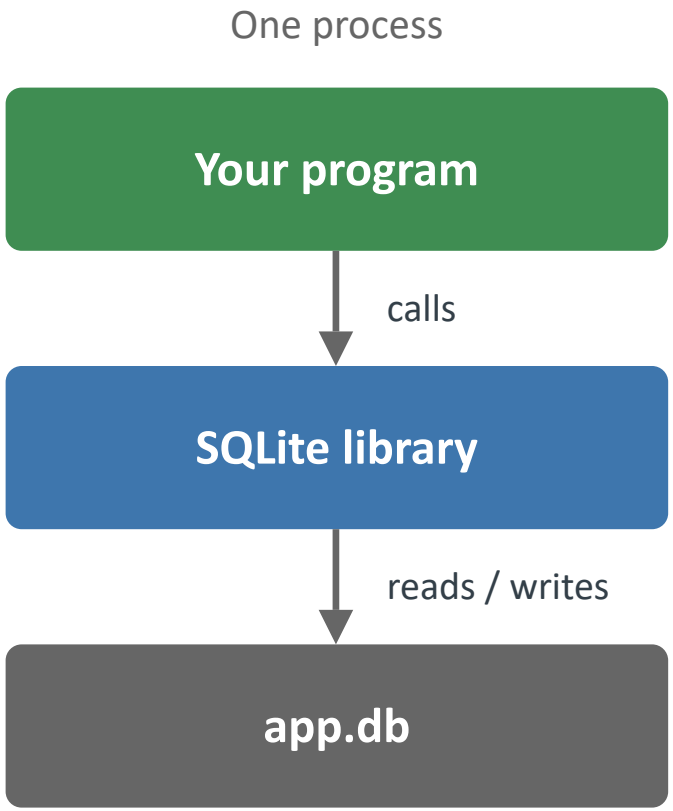
**Database**
Structured records: search, filter, relate

Choose the storage model from the operations the program must support.

# SQLite is a database inside your process

No server, account, daemon, or administrator is required

- Your program calls a library instead of sending requests to a database server.
- The durable database lives in one main .db file.
- The schema, tables, indexes, and records travel together.
- SQLite may create journal or WAL sidecar files while the database is open.
- It is a strong default for local structured data and modest write concurrency.

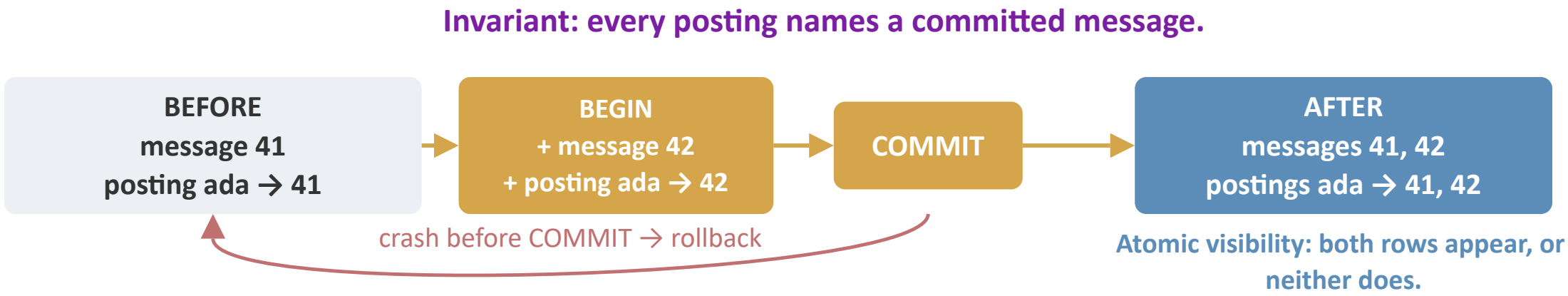



source: https://sqlite.org/serverless.html; https://sqlite.org/tempfiles.html

# A transaction is the persistence boundary

A group of changes becomes durable together or disappears together

| Moment | What other connections see | After a crash |
|---|---|---|
| Before COMMIT | the last committed state | the incomplete transaction is rolled back |
| COMMIT succeeds | all changes appear together | the committed state remains |
| A statement fails | no half-finished public state | ROLLBACK restores the prior transaction state |




source: https://sqlite.org/transactional.html; https://sqlite.org/atomiccommit.html

# Why not just a file?

- Saving one object is easy: serialize it and write a file.
- Real applications manage many records: contacts, messages, songs.
- Find one record among thousands? A flat file usually requires a scan.
- Delete one record? The application may have to rewrite the file.
- Relate records? The application invents conventions and must enforce them.
- Update two related facts? The application must invent atomic commit.

A database provides indexed lookup, structured relationships, and atomic operations without rebuilding them in every program.

# Structured data lives in tables

Rows are records; columns are attributes shared by every record

| id | name | email | year |
|---|---|---|---|
| 1 | Bart | bart@fox.com | 4 |
| 2 | Lisa | lisa@fox.com | 2 |
| 3 | Milhouse | milhouse@fox.com | 4 |
| 4 | Ralph | ralph@fox.com | 2 |

**Rows**

- One row = one student
- The row count grows with the data

**Columns**

- One column = one attribute
- The schema fixes the set; STRICT can enforce storage type

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

# A variable-length list needs its own table

Adding columns for item 1, item 2, item 3 eventually fails

- One scalar value per record fits naturally in a column.
- A playlist can contain 5, 50, or 500 songs.
- A fixed set of song1, song2, … columns cannot represent that cleanly.

| id | name | song1 | song2 | song3 | … |
|---|---|---|---|---|---|
| 1 | Road Trip | Peaches | Blue in Green | Porkpie Hat | ??? |
| 2 | Study | La Fille | | | |

If adding data requires adding columns, the schema is modeling the list incorrectly.

# The list becomes rows in a second table

Add 5 or 500 songs without changing the schema

playlists

| id | name |
|---|---|
| 1 | Road Trip |
| 2 | Study |

playlist_songs

| id | playlist_id | title | pos |
|---|---|---|---|
| 1 | 1 | Peaches En Regalia | 1 |
| 2 | 1 | Blue in Green | 2 |
| 3 | 1 | Goodbye Porkpie Hat | 3 |
| 4 | 2 | La Fille Aux Cheveux | 1 |

playlist_id links each song to its playlist; pos preserves order inside the list.

# A primary key gives each row one identity

The database rejects duplicates, and other tables can refer to the row

students

| id (PK) | name | email |
|---|---|---|
| 1 | Bart | bart@fox.com |
| 2 | Lisa | lisa@fox.com |
| 3 | Milhouse | milhouse@fox.com |

**Declare and assign the key**

```
CREATE TABLE students (
  id    INTEGER PRIMARY KEY,
  name  TEXT NOT NULL,
  email TEXT UNIQUE
) STRICT;

INSERT INTO students(name, email)
VALUES ('Nelson', 'nelson@fox.com');
-- id omitted: SQLite chooses one
```

- A table has at most one primary key; it may contain one column or several.
- INTEGER PRIMARY KEY is SQLite's row identity. If an INSERT omits it, SQLite chooses a unique integer.
- A duplicate key is rejected. Names and email addresses make poor primary keys because they can change.

source: https://sqlite.org/lang_createtable.html#the_primary_key; https://sqlite.org/lang_createtable.html#rowid

# A foreign key requires a matching parent

The child stores a value; the constraint gives that value meaning

**Parent table**

```
CREATE TABLE playlists (
  id   INTEGER PRIMARY KEY,
  name TEXT NOT NULL
) STRICT;
```

**Enable enforcement; declare the child**

```
PRAGMA foreign_keys = ON;

CREATE TABLE playlist_songs (
  id          INTEGER PRIMARY KEY,
  playlist_id INTEGER NOT NULL
    REFERENCES playlists(id)
    ON DELETE CASCADE,
  title       TEXT NOT NULL
) STRICT;
```

- playlist_songs.playlist_id points to playlists.id; inserting 99 fails if playlist 99 does not exist.
- The parent column must be a primary key or otherwise UNIQUE.
- NOT NULL makes the relationship required; without it, NULL means ‘no parent.’
- SQLite requires PRAGMA foreign_keys = ON for every database connection.

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

# Delete behavior is part of the relationship

A foreign key can protect children or propagate a parent's deletion

| Declaration | DELETE playlists.id = 1 | Use when |
|---|---|---|
| default: NO ACTION | rejected while child rows remain | children must be handled first |
| ON DELETE CASCADE | matching child rows are deleted | children belong to the parent |
| ON DELETE SET NULL | child key becomes NULL | children may outlive the parent |

- CASCADE can delete many rows, so choose it deliberately.
- SET NULL requires a nullable child column; it conflicts with playlist_id INTEGER NOT NULL.
- The same idea applies to parent-key changes through ON UPDATE actions.

The schema decides whether a parent change is forbidden or propagated.

source: https://sqlite.org/foreignkeys.html#fk_actions

# Two foreign keys can form one composite primary key

A junction table stores a many-to-many relationship as rows

students

| id | name |
|---|---|
| 1 | Bart |
| 2 | Lisa |
| 3 | Milhouse |

courses

| id | name | teacher |
|---|---|---|
| 101 | CS 371M | Krabappel |
| 102 | Math 301 | Hoover |

enrollments

| student_id (PK, FK) | course_id (PK, FK) | grade |
|---|---|---|
| 1 | 101 | B- |
| 1 | 102 | C |
| 2 | 101 | A+ |
| 3 | 102 | B+ |

PRIMARY KEY (student_id, course_id) — the pair rejects duplicate enrollments.

# Instapoll

Instapoll · Review · Sep 3 B

Your program stores thousands of messages that must survive restarts and be filtered by sender and date. What most strongly favors SQLite over one flat file?

- A. Structured records, indexed queries, and atomic updates in a local database
- B. A flat file always uses less disk space than any database representation
- C. SQLite removes the need to decide what fields each message contains
- D. A database guarantees that every query runs in constant time

# Key Takeaways

1. SQLite gives a program structured local persistence without running a database server.
2. Tables, primary keys, foreign keys, and constraints make data relationships explicit and checkable.
3. A committed change survives a crash; W4-Tue’s companion deck writes the SQL this schema needs.

# Design one durable notes database

10 min

- Model notes with a title and body; each note may have any number of tags.
- Example note: title: Truckin · body: “Arrows of neon and flashing marquees out on main street” · tags: shuffle, bluesy, lots-of-lyrics.
- Sketch the two tables, including primary keys and the foreign key that links a tag to its note.
- Decide what should happen to a note’s tags when the note is deleted, and declare it.
- Circle the changes that must commit together when a new note and its tags are inserted.
- Deliverable: the schema, the delete behavior, and the transaction boundary on one page.