```
sqlite3 pinewick.db
```

```
sqlite> SELECT m.name, round(sum(t.distance_km), 1) AS km
   ...> FROM signups s JOIN members m USING (member_id)
   ...> JOIN hikes h USING (hike_id) JOIN trails t USING (trail_id)
   ...> WHERE s.attended = 1
   ...> GROUP BY m.name ORDER BY km DESC;
```

name	km
Sam Patel	37.9
Lena Fischer	26.7
Maya Ortiz	16.6
Theo Grant	13.2



```
pinewick.db · one file · no server · ACID transactions
sqlite> PRAGMA journal_mode = WAL; → wal
```

FROM FIRST QUERY TO PRODUCTION

SQLite for Developers

Production-ready embedded databases for apps, tools, and data

Every SQL example run on SQLite 3.42 · Python · C# · mobile notes

SQL · JSON · FTS5 · WAL · INDEXES · MIGRATIONS

THENEXBOOKS

SQLite
for Developers

From Your First Query to Production-Ready Embedded Databases

Queries · Schema Design · JSON · Full-Text Search · Performance · Apps

First Edition

THENEXBOOKS

SQLite for Developers
From Your First Query to Production-Ready Embedded Databases

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. SQLite features vary by version and by how a particular copy was compiled; always consult the official documentation at sqlite.org for the version you use. Test every change on a copy of your data first.

SQLite is in the public domain; the name "SQLite" is used here for identification only. Python is a trademark of the Python Software Foundation. Microsoft, Windows, Visual Studio Code, and .NET are trademarks of the Microsoft group of companies. Java is a registered trademark of Oracle and/or its affiliates. Android is a trademark of Google LLC. iOS is a trademark or registered trademark of Apple Inc. Other product names are trademarks of their respective owners. This book is independent and is not affiliated with, sponsored by, or endorsed by the SQLite developers or any company named in it.

The Pinewick Trail Club and its Trailbook app, the running example in this book, are fictional, as are all members, trails, hikes, and coordinates in its examples. Any resemblance to real clubs, places, or people is coincidental.

Preface

SQLite is the database you probably already use without noticing. It lives inside phones, browsers, desktop applications, command-line tools, and embedded devices, quietly storing settings, caches, messages, and business data in a single ordinary file. There is no server to install, no user accounts to configure, and no network port to secure. You link a small library into your program, open a file, and you have a full SQL database with transactions, indexes, triggers, views, JSON support, and full-text search.

That simplicity is also why SQLite is so often used carelessly. Developers treat it as a "toy" database, never turn on foreign keys, store dates in three different formats, open a new connection for every query, hit "database is locked" under load, and copy a live database file as a backup. None of these problems are SQLite's fault. They come from not knowing how it works. This book aims to give you that knowledge.

We start from the very first `SELECT` in the `sqlite3` shell and build steadily: filtering and sorting, SQLite's flexible type system, built-in functions for text and dates, joins, aggregation, CTEs, and window functions. Then we design tables with constraints that protect your data, change data safely with transactions, add views and triggers, and evolve schemas with migrations. We go beyond plain tables with JSON, full-text search, and spatial indexes, then make databases fast and reliable with indexes, query plans, WAL mode, backups, and security practices. Finally, we use SQLite from Python and other languages, test it, deploy it, and build a complete offline-first application. Every example was actually run: SQL on SQLite 3.42, shell dot-commands in the sqlite3 shell, and application code in Python. The results you see are the results those runs produced.

Who This Book Is For

- **Beginners** who want a complete, practical path from first query to professional use of SQL.
- **Application developers** building mobile, desktop, web, or command-line apps that need a reliable local database.
- **Data analysts and scientists** who want a fast, file-based SQL engine for exploring and sharing data.
- **Developers who already use SQLite** and want to understand locking, typing, performance, and production practices properly.

What You Will Learn

- How to install the `sqlite3` shell and GUI tools, and create the lab database used throughout the book.
- Querying fundamentals, storage classes and type affinity, STRICT tables, NULL logic, and built-in functions for text, numbers, dates, and times.

- Joins, grouping, subqueries, common table expressions (including recursive ones), window functions, and set operations.
- Table design with keys and constraints; `INSERT`, `UPDATE`, `DELETE`, UPSERT, and `RETURNING`.
- Transactions and savepoints, views, triggers, generated columns, and safe schema migrations.
- JSON functions, full-text search with FTS5, R*Tree spatial indexes, and importing, exporting, and attaching databases.
- Index design, `EXPLAIN QUERY PLAN`, WAL mode and concurrency, backups and integrity checks, and security.
- Using SQLite from Python, C#, Java, Node.js, Go, and mobile platforms; testing, migrations, and deployment.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) introduces SQLite, sets up the tools, and teaches querying, types, and functions. **Part II — Querying in Depth** (Chapters 6–10) covers joins, aggregation, CTEs, window functions, and advanced expressions. **Part III — Designing and Changing Data** (Chapters 11–15) builds tables, modifies data safely, and evolves schemas. **Part IV — Beyond Plain Tables** (Chapters 16–19) adds JSON, full-text search, virtual tables, and data exchange. **Part V — Performance and Reliability** (Chapters 20–24) makes databases fast, concurrent, recoverable, and secure. **Part VI — SQLite in Applications** (Chapters 25–28) puts SQLite to work inside real programs and ends with a complete case study. The appendices provide a cheat sheet, a function, PRAGMA, and template reference, a production checklist, and a glossary.

Each chapter ends with practice exercises and a **Key Takeaways** box for quick review.

The Running Example: Pinewick Trail Club

Throughout the book we work with the database of the **Pinewick Trail Club**, a fictional hiking club, and its offline-first app, **Trailbook**. The club has members, trails in three regions, organized hikes, and sign-ups with attendance and ratings. The data includes a member with no email address, a member who has never signed up, a trail nobody has hiked yet, a cancelled hike, and a planned hike whose results are not known. It is small enough to check every result by eye, yet it exercises everything a real application needs. Chapter 2 gives you the complete script to create it, so you can run every example yourself.

Versions and Conventions

SQL examples were run on SQLite 3.42 and work on that version or any later one; a few shell sessions were captured with a newer sqlite3 shell, as Chapter 2 explains. When a feature appeared in a specific version, we say so, and features newer than 3.42 are labeled and described without output. Shell

sessions are shown with the `sqlite>` prompt and the shell's `box` output mode, with `NULL` displayed as the word NULL (Chapter 2 shows how to set this up). Code appears in `monospace`.

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, version details, or a useful aside.

> **WARNING**
>
> A common mistake that causes wrong results, data loss, slow queries, or security holes.

Contents

Preface

PART I — FOUNDATIONS

1 What SQLite Is and When to Use It

2 Setting Up: The sqlite3 Shell, GUI Tools, and the Pinewick Database

3 Querying Basics: SELECT, WHERE, ORDER BY, and LIMIT

4 Storage Classes, Type Affinity, NULLs, and STRICT Tables

5 Built-in Functions: Text, Numbers, Dates, and Times

PART II — QUERYING IN DEPTH

6 Joins: INNER, LEFT, RIGHT, FULL, CROSS, and Self Joins

7 Grouping and Aggregation

8 Subqueries and Common Table Expressions

9 Window Functions

10 CASE, Set Operations, and Advanced Expressions

PART III — DESIGNING AND CHANGING DATA

11 Creating Tables: Keys, Constraints, ROWID, and AUTOINCREMENT

12 INSERT, UPDATE, DELETE, UPSERT, and RETURNING

13 Transactions, Savepoints, and Journal Modes

14 Views, Triggers, and Generated Columns

15 Changing a Schema Safely: ALTER TABLE, Table Rebuilds, and Migrations

PART IV — BEYOND PLAIN TABLES

16 JSON in SQLite

17 Full-Text Search with FTS5

18 R*Tree Indexes, Virtual Tables, and Extensions

19 Importing, Exporting, and Attaching Databases

PART V — PERFORMANCE AND RELIABILITY

20 Indexes: How They Work and How to Design Them

21 EXPLAIN QUERY PLAN and Query Tuning

22 Concurrency: WAL Mode, Busy Timeouts, and One Writer at a Time

23 Backup, VACUUM, Integrity Checks, and Recovery

24 Security: SQL Injection, Untrusted Databases, and Encryption Options

PART VI — SQLITE IN APPLICATIONS

25 SQLite from Python

26 SQLite from Other Languages: C#, Java, Node.js, Go, and Mobile

27 Testing, Migrations, and Deployment for SQLite Apps

28 Case Study: Trailbook, an Offline-First Club App

APPENDICES

A SQLite Cheat Sheet

B Function, PRAGMA, and Template Reference

C Production Checklist

D Glossary

PART I

Foundations

What SQLite is and when to use it, the sqlite3 shell and the Pinewick lab database, and the fundamentals of querying, SQLite's flexible type system, NULLs, and built-in functions.

1 · What SQLite Is

2 · Setting Up

3 · Querying Basics

4 · Types, NULLs, and STRICT Tables

5 · Built-in Functions

CHAPTER 1 · PART I — FOUNDATIONS

What SQLite Is and When to Use It

SQLite is a complete SQL database engine that fits inside your program and keeps everything in one ordinary file. This chapter explains what that means, where SQLite shines, where a client/server database is the better choice, and how the rest of the book is organized around a small hiking club and its app.

A Database Engine in a Library

Most developers meet databases as **servers**: separate programs such as PostgreSQL, MySQL, or SQL Server that run in the background, listen on a network port, and answer queries sent to them by client programs. Your application opens a connection, sends SQL text over a socket, and waits for rows to come back. Someone has to install that server, start it, give it users and passwords, back it up, and keep it running.

SQLite takes a completely different approach. It is a **library**, a body of C code that you link into your own program (or that your language runtime already includes). When your code runs a query, there is no network round trip and no second process. The library reads and writes the database file directly, inside your process, using ordinary file system calls. The SQLite documentation describes this with a memorable phrase: SQLite does not compete with client/server databases; it competes with `fopen()`. In other words, the honest comparison is not "SQLite or PostgreSQL?" but "SQLite or my own home-made file format?"

Despite living in a library, SQLite is a real relational database engine. It parses and plans SQL, supports tables, indexes, views, triggers, joins, subqueries, common table expressions, window functions, JSON, and full-text search, and it protects your data with **transactions**: groups of changes that either happen completely or not at all, even if the power fails halfway through. You will use every one of those features in this book.

Embedded

We say SQLite is **embedded** because it runs embedded inside the host application. Python ships it in the standard library as the `sqlite3` module. Android and iOS include it in the operating system. Many other languages have a driver that bundles a copy of the library. The consequence is that "installing SQLite" for an application usually means doing nothing at all: the engine is already there.

Here is a complete Python program that opens the club database you will build in Chapter 2 and runs one query. Notice that there is no host name, no port, no user name, and no password. There is only a file name.

hello.py

```
import sqlite3

conn = sqlite3.connect("pinewick.db")      # opens (or creates) one file
for name, km in conn.execute(
        "SELECT name, distance_km FROM trails WHERE region = ?", ("Lakeside",)):
    print(name, km)
conn.close()
```

Output

```
Mirror Lake Path 7.8
Quarry Steps 9.6
```

The `?` in the query is a **parameter**: a placeholder that the library fills in safely with the value `"Lakeside"`. Parameters are the single most important habit for application code, and Chapter 24 explains why they are your main defense against SQL injection. Chapter 25 covers the Python module in depth.

Serverless and zero configuration

Because there is no server process, SQLite is called **serverless**. (The word predates the cloud meaning of "serverless functions"; here it simply means there is no database server to run.) It is also **zero configuration**: there is no configuration file to edit, no service to register, no port to open in a firewall, and no administrator account to create. A program that wants a database simply opens a file name. If the file does not exist, SQLite creates it.

That does not mean SQLite has no settings. It has many, called **pragmas**, and you will meet important ones throughout the book, such as `PRAGMA foreign_keys` in Chapter 2 and `PRAGMA journal_mode` in Chapters 13 and 22. The difference is that pragmas are set by the application on its own connection, in SQL, rather than by an administrator on a shared server.

One database, one file

A SQLite database is a **single file** on disk. The tables, the indexes, the views, the triggers, and the schema itself all live inside it. You can copy that file to a USB stick, attach it to an email, check it into a test fixture folder, or open it on a different operating system, and it just works: the file format is **cross-platform**, so a database written on a 64-bit Linux server can be read by a 32-bit Windows program or a phone, regardless of byte order or word size. The format is also stable and documented, and the SQLite developers have committed to keeping it readable by future versions for the long term.

Inside the file, data is organized in fixed-size **pages**. The default page size is 4096 bytes. When you build the Pinewick database in Chapter 2, it fits in nine pages:

sqlite3

```
sqlite> PRAGMA page_size;
┌───────────┐
│ page_size │
├───────────┤
│ 4096      │
└───────────┘
sqlite> PRAGMA page_count;
┌────────────┐
│ page_count │
├────────────┤
│ 9          │
└────────────┘
```

Nine pages of 4096 bytes is 36,864 bytes, which is exactly the size the operating system reports for `pinewick.db`. The database can grow much larger: the default maximum database size is very large (terabytes), far beyond what most embedded uses need.

> **NOTE**
>
> While a database is being written, SQLite may create a temporary companion file next to it: a `-journal` file in the default rollback mode, or `-wal` and `-shm` files in WAL mode. These files are part of how SQLite keeps your data safe during a crash. Never delete them by hand while the database might be in use. Chapter 13 explains journal modes, and Chapter 23 explains how to make a correct backup copy of a live database.

Public domain

SQLite's source code is in the **public domain**. That is unusual even among open-source projects: there is no license to comply with, no attribution requirement, and no copyleft obligation. You can use it in a commercial product, modify it, and ship it without asking anyone. For organizations that need a formal legal document anyway, the SQLite developers offer a purchasable warranty of title, but for most developers the practical meaning is simple: you are free to use it anywhere.

The project is known for its extremely thorough testing and careful backward compatibility.

A forgiving SQL dialect

SQLite speaks a large subset of standard SQL, plus some extensions of its own. If you know SQL from another database, most queries will look familiar. The biggest difference is the type system. In most databases, a column declared as `INTEGER` can only hold integers. In SQLite, the declared type is a preference, called **type affinity**, and by default a value of a different type can still be stored. That flexibility is convenient but has sharp edges, so Chapter 4 explains it in detail and shows how **STRICT tables** give you conventional, rigid typing when you want it.

Where SQLite Is Used

SQLite is one of the most widely deployed database engines in the world. You almost certainly have many copies of it on the devices around you right now. We will not quote numbers, because they change constantly and are hard to verify, but the kinds of places it appears are easy to describe.

- **Mobile apps.** Android and iOS both include SQLite, and it is the usual foundation for on-device storage. Frameworks such as Room on Android and Core Data on iOS (which can use SQLite as a store) sit on top of it. Chapter 26 discusses mobile use.
- **Web browsers and desktop applications.** Browsers, email clients, music players, photo managers, and note-taking apps commonly keep bookmarks, history, caches, and libraries in SQLite files.
- **Embedded devices.** Cars, set-top boxes, cameras, routers, and other devices with modest hardware use SQLite because it is small, needs no administration, and survives sudden power loss.
- **Application file formats.** Instead of inventing a custom binary format, many programs save their documents as SQLite databases. The document becomes queryable, can be updated in place without rewriting the whole file, and is protected by transactions.
- **Data analysis and data exchange.** A SQLite file is a convenient way to ship a data set: one file that the recipient can query with SQL immediately. Chapter 19 covers importing CSV and JSON and attaching several database files.
- **Testing and prototyping.** Because a SQLite database can live entirely in memory, test suites often use it for fast, isolated tests. Chapter 27 shows the practice and its limits.
- **Websites and services.** Many low- and medium-traffic websites run happily on SQLite, especially read-heavy ones. Modern hardware, WAL mode (Chapter 22), and careful indexing (Chapter 20) take it a long way.

What these uses have in common is **locality**: the data lives on the same machine as the code that uses it, and the program that owns the data is usually the only one writing to it. That is the sweet spot for an embedded database.

When to Use SQLite

Choosing a database is an engineering decision, not a matter of loyalty. The following situations are strong signals that SQLite is the right tool.

- **The data belongs to one application on one machine.** A desktop tool, a mobile app, a command-line utility, or a single web server process. SQLite is simpler and faster here than any client/server system because there is no network hop.
- **You need offline operation.** An app that must work in a forest with no signal, like the Trailbook app in this book, needs its data on the device. SQLite gives it a real database there.

- **You want reliable structured storage instead of ad-hoc files.** If you are tempted to write JSON or CSV files and rewrite them on every change, SQLite gives you atomic updates, indexes, and queries for less code.
- **You need a portable, self-contained data set.** One file is easy to distribute, archive, and open with free tools.
- **Reads greatly outnumber writes,** or writes arrive at a rate one writer can handle. SQLite allows many simultaneous readers but only one writer at a time, and each write transaction is typically very short.
- **Operational simplicity matters.** No server means nothing to patch, monitor, or restart, and backups can be as simple as calling the backup API (Chapter 23).
- **You are learning SQL.** SQLite is an excellent place to learn: nothing to install beyond one small program, instant feedback, and you can delete the file and start over at any time.

> **TIP**
> A useful rule of thumb from the SQLite documentation: if many separate client machines need to reach the same data over a network, choose a client/server database. If the data and the program live together, SQLite is usually the better answer.

When to Choose a Client/Server Database

SQLite's design gives up some things on purpose. When your project needs them, choose a client/server database such as PostgreSQL, MySQL, MariaDB, or SQL Server instead.

- **Many machines writing to the same data.** If application servers on several hosts must all read and write one shared database, a server that they connect to over the network is the natural design. SQLite works on files, and sharing a database file over a network file system is risky because many network file systems do not implement file locking correctly.
- **High write concurrency.** SQLite serializes writes: one write transaction at a time per database. For many applications that limit is never reached, but a system with a constant stream of concurrent writers from many users will be better served by a server that can write to different rows in parallel. Chapter 22 shows how far you can go with WAL mode, busy timeouts, and short write transactions.
- **Built-in users, roles, and permissions.** SQLite has no user accounts and no `GRANT` statement. Whoever can read the file can read the data, and whoever can write the file can change it. Access control is the job of the operating system and your application.
- **Very large, centrally managed data sets.** SQLite can store very large databases, but if the data must be spread across machines, replicated with automatic failover, or managed by a database team, client/server systems provide those tools out of the box.
- **Features specific to a server product.** Stored procedures, a rich set of built-in data types, or a particular extension ecosystem may tip the balance.

The following table summarizes the trade-offs.

Question	SQLite	Client/server database
Where does the engine run?	Inside your process	In a separate server process
Setup and administration	None; open a file	Install, configure, secure, monitor
Storage	One cross-platform file	A directory of files managed by the server
Concurrent writers	One at a time per database	Many, with row-level locking
Network access	No; local file access	Yes; that is its purpose
Users and permissions	None; file permissions only	Accounts, roles, GRANT/REVOKE
Typical sweet spot	Apps, devices, tools, files, small to medium sites	Shared multi-user systems

Many real systems use both. A mobile app keeps an SQLite database on the phone and synchronizes with a PostgreSQL database on a server. A data pipeline exports a nightly snapshot from a server database into an SQLite file for analysts. A web service uses a server database for its main data and SQLite files for caches or per-tenant archives. Knowing both tools well lets you use each where it fits.

> **WARNING**
>
> Do not put a SQLite database that several computers write to on a network share (SMB, NFS, or a synced cloud folder) and hope for the best. Locking on network file systems is often unreliable, and the result can be a corrupted database. If several machines need the data, use a client/server database or put a single service in front of the SQLite file.

Common misunderstandings

Because SQLite is small and free, people sometimes underestimate it. Three misunderstandings come up often enough to address now.

"SQLite is a toy for prototypes." It is mature, heavily tested software that stores critical data on a very large number of devices. The question is never whether SQLite is "serious enough" but whether its architecture (embedded, one writer at a time, local file) matches your workload.

"SQLite cannot handle concurrency." It handles concurrent *readers* very well, and in WAL mode readers do not block the writer or each other. What it does not allow is two writers at the same moment. Because write transactions are usually short, many applications never notice. Chapter 22 is devoted to this topic.

"SQLite has no data types." It has a small set of storage classes and a flexible typing system, and it also offers STRICT tables that reject wrongly typed values. Chapter 4 explains both so that the flexibility never surprises you.

How This Book Works

This book takes you from your first query to the level of a professional application developer who uses SQLite in production. Part I assumes no SQL knowledge at all. Each chapter builds on the previous ones, but later chapters are written so that you can jump to them when you need a specific topic.

The six parts

- **Part I, Foundations (Chapters 1–5):** what SQLite is, setting up the shell and the lab database, basic queries, the type system and NULLs, and the built-in functions for text, numbers, dates, and times.
- **Part II, Querying in Depth (Chapters 6–10):** joins, grouping and aggregation, subqueries and common table expressions, window functions, CASE expressions, and set operations.
- **Part III, Designing and Changing Data (Chapters 11–15):** creating tables with keys and constraints, changing data with INSERT, UPDATE, DELETE, UPSERT, and RETURNING, transactions, views, triggers, generated columns, and safe schema migrations.
- **Part IV, Beyond Plain Tables (Chapters 16–19):** JSON, full-text search with FTS5, R*Tree indexes and other virtual tables, and importing, exporting, and attaching databases.
- **Part V, Performance and Reliability (Chapters 20–24):** indexes, query plans and tuning, concurrency, backup and recovery, and security.
- **Part VI, SQLite in Applications (Chapters 25–28):** SQLite from Python and from other languages, testing and deployment, and a complete case study that ties everything together.

Versions

SQLite changes slowly and carefully, but it does add features. Every example in this book was tested with **SQLite 3.42**, and everything shown works in SQLite 3.42 or later. We recommend that you install the latest release from the official website, `sqlite.org`, as Chapter 2 shows. When we mention a feature that arrived after 3.42, we label it with the version that introduced it and do not show output for it, because we did not run it for this book.

You can check the version of the library you are using at any time with the `sqlite_version()` function:

sqlite3

```
sqlite> SELECT sqlite_version();
```

sqlite_version()
3.42.0

Keep in mind that the command-line shell and your application may use different copies of the library. Python, for example, reports the version of the SQLite library it was built with, which may be older or newer than the `sqlite3` program on your path. When a feature seems to be missing, check the version from inside the program that is failing.

Conventions

SQL keywords are written in capitals, such as `SELECT` and `WHERE`, and names of tables and columns in lower case with underscores, such as `hike_date`. SQLite itself does not care about the case of keywords, so `select` works too, but consistent capitals make queries easier to scan.

Most examples are shown as sessions in the `sqlite3` command-line shell, with the `sqlite>` prompt in front of what you type and the result underneath. We use the shell's **box** output mode and display missing values as `NULL`; Chapter 2 shows you how to set up the same display so that your screen matches the book. Long queries are wrapped over several lines for readability. Every result in this book was produced by actually running the code; when we shorten a long result, we show a `...` line and say so.

Three kinds of boxes appear in every chapter. A **Tip** is a shortcut or a good habit. A **Note** gives background or a detail that is easy to miss. A **Warning** points out a mistake that can cost you data or time. Each chapter ends with a short **Practice** section with exercises that use the same database, followed by **Key Takeaways** that summarize the chapter.

New terms are printed in **bold** where they are defined. Appendix D collects them in a glossary, and Appendix A is a one-page cheat sheet of the syntax used throughout the book.

How to read the examples

You will learn far more by typing the examples than by reading them. The lab database has fixed data, so your output should match ours exactly. If it does not, an earlier experiment probably changed the data, or your shell display settings differ: rebuild the database from the script and check the settings described in Chapter 2. Experiment freely and predict each result before you press Enter; a mistake costs nothing.

Meet the Pinewick Trail Club

Every example in this book uses the same small, realistic data set. The **Pinewick Trail Club** is a fictional hiking club. Its members walk trails in three regions near the town of Pinewick: North Ridge, Lakeside, and Southwood. Club leaders schedule group hikes on particular trails and dates, and members sign up for them. After a hike, the leader records who actually came and members can rate the outing from one to five stars.

The club keeps all of this in an app called **Trailbook**. Trailbook runs on members' phones and laptops and must work offline, because the best trailheads have no mobile signal. That makes it a natural fit for SQLite: each device has its own database file, and the app synchronizes when it is back online.

Throughout the book we build up Trailbook's database step by step, and in Part VI we write the Python code for the app itself.

The four tables

The lab database starts with four tables. Chapter 2 shows the complete script that creates them; here is an overview.

Table	One row per	Main columns
`members`	club member	`member_id`, `name`, `email`, `joined_on`, `level`
`trails`	trail the club walks	`trail_id`, `name`, `region`, `distance_km`, `elevation_m`, `difficulty`, `description`
`hikes`	scheduled group hike	`hike_id`, `trail_id`, `leader_id`, `hike_date`, `max_people`, `status`
`signups`	member signed up for a hike	`hike_id`, `member_id`, `signed_up_at`, `attended`, `rating`

The tables are connected. A hike takes place on one trail (`hikes.trail_id` refers to `trails`) and is led by one member (`hikes.leader_id` refers to `members`). A sign-up connects one hike and one member, and the pair `(hike_id, member_id)` identifies it, so the same member cannot sign up twice for the same hike. Columns that refer to a row in another table are called **foreign keys**; Chapter 11 covers them in depth, and Chapter 2 shows how to make SQLite enforce them.

The people, trails, and hikes

The club is small, which keeps every result short enough to check by eye. There are five members. Maya Ortiz and Sam Patel are leaders, Lena Fischer and Theo Grant are regular members, and Priya Nair is a starter who joined recently and has not signed up for anything yet. Theo has not given the club an email address, so his `email` is missing. Missing values like that one are called **NULL** in SQL, and they behave in surprising ways that Chapter 4 explains.

There are six trails, two in each region, ranging from the 3.2 km Bramble Hill to the 13.5 km Owl Creek Trail. Each has a difficulty of `easy`, `moderate`, or `hard`. The club has scheduled six hikes between March and June 2026: four are done, one was cancelled, and one is still planned. Bramble Hill has never been hiked by the club, which is useful when you want to find things that have no matching rows.

There are 18 sign-ups. For hikes that have taken place, `attended` is 1 if the member came and 0 if they did not; for the cancelled and planned hikes it is still unknown, which means NULL. Ratings are whole numbers from 1 to 5, or NULL when the member gave no rating.

Here is a first taste of what you will be able to do by the end of Part II: a single query that combines three tables to show who is leading the next planned hike, and where.

sqlite3

```
sqlite> SELECT m.name AS leader, h.hike_date, t.name AS trail
   ...> FROM hikes AS h
   ...> JOIN members AS m ON m.member_id = h.leader_id
   ...> JOIN trails  AS t ON t.trail_id  = h.trail_id
   ...> WHERE h.status = 'planned';
```

leader	hike_date	trail
Maya Ortiz	2026-06-13	Mirror Lake Path

Do not worry about the details yet. `JOIN` is the subject of Chapter 6. The `...>` prompt is how the shell shows that a statement continues on the next line; it waits for the semicolon before running anything.

And here is a quick count of every table, so that you know what "correct" looks like once you have built the database in the next chapter:

sqlite3

```
sqlite> SELECT (SELECT count(*) FROM members) AS members,
   ...>        (SELECT count(*) FROM trails)  AS trails,
   ...>        (SELECT count(*) FROM hikes)   AS hikes,
   ...>        (SELECT count(*) FROM signups) AS signups;
```

members	trails	hikes	signups
5	6	6	18

How the example grows

The four tables are only the beginning. As the book progresses, Trailbook gains new pieces, each introduced in the chapter that teaches the feature behind it:

- a view that summarizes each hike, triggers that log status changes and stop a full hike from accepting more sign-ups, and a generated column that measures how steep a trail is (Chapter 14);
- a numbered list of schema migrations, tracked inside the database file itself (Chapter 15);
- members' gear lists stored as JSON (Chapter 16), full-text search over trail descriptions (Chapter 17), and a spatial index of trailheads (Chapter 18);
- a large table of generated check-ins for performance experiments (Chapters 20 and 21);
- and finally a small Python package, `trailbook`, with a command-line interface and a test suite (Chapters 25, 27, and 28).

Because the data is fictional and small, you can always predict what a query should return. That is deliberate: when you are learning a new feature, it helps enormously to know the right answer before you run the query.

> **NOTE**
>
> All names, email addresses, places, and coordinates in the Pinewick data are invented. Email addresses use the reserved `.example` domain so that they can never reach a real person.

What You Need

To follow along you need a computer running Windows, macOS, or Linux, a text editor, and the `sqlite3` command-line shell, which Chapter 2 helps you install. A graphical tool is optional but pleasant for browsing data. For Part VI you will also want Python 3; the examples use only its standard library. No server and no account are required.

You do not need prior SQL experience. If you already know SQL from another database, you can move quickly through Chapter 3, but read Chapter 4 carefully: the type system and the treatment of NULLs are where SQLite most often surprises experienced developers.

Practice

1. List three programs or devices you use every week that are likely to store data locally. For each one, decide whether an embedded database or a client/server database would be the better fit, and write one sentence explaining why.
2. A team is building a booking system where staff in five offices update the same calendar at the same time over the company network. Would you choose SQLite or a client/server database? Name the two characteristics of the workload that drive your decision.
3. A field researcher needs an app that records bird sightings on a tablet with no connectivity for weeks at a time. Explain why SQLite is a good fit and what has to happen when the tablet returns to the office.
4. Using only the description in this chapter, predict how many members have no sign-ups and how many trails have never been hiked. You will check your answers with queries in later chapters.