Small. Fast. Reliable. Choose Any Three.

Using

SQLite

O'REILLY®

Jay A. Kreibich

Using SQLite

Application developers, take note: databases aren't just for the IS group any more. You can build database-backed applications for the desktop, Web, embedded systems, or operating systems without linking to heavy-duty client-server databases such as Oracle and MySQL. This book shows you how to use SQLite, a small and lightweight relational database engine that you can build directly into your application.

With SQLite, you can develop a database-backed application that remains manageable both in size and complexity. This book guides you every step of the way. You'll get a crash course in data modeling, become familiar with SQLite's dialect of the SQL database language, and much more.

- Learn how to maintain localized storage in a single file that requires no configuration
- Build your own SQLite library or use a precompiled distribution in your application
- Get a primer on SQL, and learn how to use several language functions and extensions
- Work with SQLite using a scripting language or a C-based language such as C# or Objective-C
- Understand the basics of database design, and learn how to transfer what you already know to SQLite
- Take advantage of virtual tables and modules

"Complex SQL concepts explained clearly."

—D. Richard Hipp
creator of SQLite

Jay A. Kreibich is a professional software engineer who has always been interested in how people process and understand information. He currently works for Volition, Inc., a software studio that specializes in open-world video games.

Previous programming experience is recommended.

O'REILLY®
oreilly.com

US $49.99 CAN $57.99
ISBN: 978-0-596-52118-9

Safari Books Online
Free online edition
for 45 days with purchase of this book. Details on last page.

Using SQLite

Using SQLite

Jay A. Kreibich

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

Using SQLite
by Jay A. Kreibich

Copyright © 2010 Jay A. Kreibich. All rights reserved.
Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editor: Mike Loukides
Production Editor: Kristen Borg
Proofreader: Kiel Van Horn
Indexer: Lucie Haskins
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

Printing History:
August 2010: First Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Using SQLite*, the image of a great white heron, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-0-596-52118-9

[M]

1281104401

To my Great-Uncle Albert "Unken Al" Kreibich.

1918–1994

He took a young boy whose favorite question was "why?" and taught him to ask the question "how?"

(Who also—much to the dismay of his parents and the kitchen telephone—taught him the joy of answering that question, especially if it involved pliers or screwdrivers.)

—jk

Table of Contents

Preface xv

1. What Is SQLite? 1
Self-Contained, No Server Required 2
Single File Database 4
Zero Configuration 4
Embedded Device Support 5
Unique Features 5
Compatible License 6
Highly Reliable 6

2. Uses of SQLite 9
Database Junior 9
Application Files 10
Application Cache 11
Archives and Data Stores 11
Client/Server Stand-in 11
Teaching Tool 12
Generic SQL Engine 13
Not the Best Choice 13
Big Name Users 15

3. Building and Installing SQLite 17
SQLite Products 17
Precompiled Distributions 18
Documentation Distribution 18
Source Distributions 19
The Amalgamation 19
Source Files 19
Source Downloads 20
Building 21

Configure 21
Manually 22
Build Customization 23
Build and Installation Options 23
An sqlite3 Primer 24
Summary 26

4. The SQL Language 27
Learning SQL 27
Brief Background 28
Declarative 28
Portability 29
General Syntax 30
Basic Syntax 30
Three-Valued Logic 31
Simple Operators 33
SQL Data Languages 34
Data Definition Language 34
Tables 35
Views 43
Indexes 44
Data Manipulation Language 45
Row Modification Commands 46
The Query Command 49
Transaction Control Language 51
ACID Transactions 51
SQL Transactions 53
Save-Points 55
System Catalogs 57
Wrap-up 58

5. The SELECT Command 61
SQL Tables 61
The SELECT Pipeline 62
FROM Clause 63
WHERE Clause 68
GROUP BY Clause 69
SELECT Header 70
HAVING Clause 73
DISTINCT Keyword 74
ORDER BY Clause 74
LIMIT and OFFSET Clauses 75
Advanced Techniques 76

Subqueries 76
Compound SELECT Statements 77
Alternate JOIN Notation 78
SELECT Examples 79
Simple SELECTs 80
Simple JOINs 80
JOIN...ON 81
JOIN...USING, NATURAL JOIN 82
OUTER JOIN 82
Compound JOIN 82
Self JOIN 83
WHERE Examples 83
GROUP BY Examples 84
ORDER BY Examples 85
What's Next 85

6. Database Design 87
Tables and Keys 87
Keys Define the Table 87
Foreign Keys 89
Foreign Key Constraints 90
Generic ID Keys 91
Keep It Specific 92
Common Structures and Relationships 93
One-to-One Relationships 93
One-to-Many Relationships 95
Many-to-Many Relationships 97
Hierarchies and Trees 99
Normal Form 102
Normalization 103
Denormalization 103
The First Normal Form 104
The Second Normal Form 104
The Third Normal Form 105
Higher Normal Forms 106
Indexes 107
How They Work 107
Must Be Diverse 108
INTEGER PRIMARY KEYs 109
Order Matters 109
One at a Time 110
Index Summary 111
Transferring Design Experience 112

Tables Are Types 112
Keys Are Backwards Pointers 113
Do One Thing 113
Closing 114

7. **C Programming Interface** **115**
API Overview 115
Structure 116
Strings and Unicode 117
Error Codes 118
Structures and Allocations 118
More Info 119
Library Initialization 119
Database Connections 120
Opening 120
Special Cases 121
Closing 122
Example 122
Prepared Statements 123
Statement Life Cycle 123
Prepare 124
Step 126
Result Columns 127
Reset and Finalize 130
Statement Transitions 131
Examples 132
Bound Parameters 133
Parameter Tokens 133
Binding Values 135
Security and Performance 138
Example 140
Potential Pitfalls 141
Convenience Functions 142
Result Codes and Error Codes 146
Standard Codes 146
Extended Codes 148
Error Functions 148
Prepare v2 149
Transactions and Errors 150
Database Locking 151
Utility Functions 156
Version Management 156
Memory Management 157

Summary 158

8. **Additional Features and APIs** **159**
Date and Time Features 159
Application Requirements 160
Representations 160
Time and Date Functions 162
ICU Internationalization Extension 167
Full-Text Search Module 169
Creating and Populating FTS Tables 169
Searching FTS Tables 170
More Details 171
R*Trees and Spatial Indexing Module 171
Scripting Languages and Other Interfaces 172
Perl 172
PHP 173
Python 173
Java 174
Tcl 174
ODBC 175
.NET 175
C++ 175
Other Languages 176
Mobile and Embedded Development 176
Memory 176
Storage 177
Other Resources 178
iPhone Support 178
Other Environments 179
Additional Extensions 180

9. **SQL Functions and Extensions** **181**
Scalar Functions 182
Registering Functions 182
Extracting Parameters 184
Returning Results and Errors 186
Example 189
Aggregate Functions 194
Defining Aggregates 194
Aggregate Context 195
Example 197
Collation Functions 200
Registering a Collation 201

Collation Example 202
SQLite Extensions 204
Extension Architecture 205
Extension Design 206
Example Extension: sql_trig 207
Building and Integrating Static Extensions 209
Using Loadable Extensions 211
Building Loadable Extensions 212
Loadable Extension Security 213
Loading Loadable Extensions 213
Multiple Entry Points 215
Chapter Summary 215

10. Virtual Tables and Modules 217
Introduction to Modules 218
Internal Modules 218
External Modules 218
Example Modules 219
SQL for Anything 219
Module API 220
Simple Example: dblist Module 224
Create and Connect 224
Disconnect and Destroy 229
Query Optimization 230
Custom Functions 231
Table Rename 232
Opening and Closing Table Cursors 233
Filtering Rows 235
Extracting and Returning Data 237
Virtual Table Modifications 239
Cursor Sequence 240
Transaction Control 241
Register the Module 243
Example Usage 245
Advanced Example: weblog Module 246
Create and Connect 248
Disconnect and Destroy 249
Other Table Functions 250
Open and Close 250
Filter 252
Rows and Columns 254
Register the Module 259
Example Usage 259

Best Index and Filter 262
Purpose and Need 262
xBestIndex() 263
xFilter() 266
Typical Usage 267
Wrap-Up 268

A. SQLite Build Options ... 269

B. sqlite3 Command Reference ... 287

C. SQLite SQL Command Reference ... 299

D. SQLite SQL Expression Reference ... 341

E. SQLite SQL Function Reference ... 361

F. SQLite SQL PRAGMA Reference ... 381

G. SQLite C API Reference ... 409

Index ... 491

Preface

This book provides an introduction to the SQLite database product. SQLite is a zero-configuration, standalone, relational database engine that is designed to be embedded directly into an application. Database instances are self-contained within a single file, allowing easy transport and simple setup.

Using SQLite is primarily written for experienced software developers that have never had a particular need to learn about relational databases. For one reason or another, you now find yourself with a large data management task, and are hoping a product like SQLite may provide the answer. To help you out, the various chapters cover the SQL language, the SQLite C programming API, and the basics of relational database design, giving you everything you need to successfully integrate SQLite into your applications and development work.

The book is divided into two major sections. The first part is a traditional set of chapters that are primarily designed to be read in order. The first two chapters provide an in-depth look at exactly what SQLite provides and how it can be used. The third chapter covers downloading and building the library. Chapters Four and Five provide an introduction to the SQL language, while Chapter Six covers database design concepts. Chapter Seven covers the basics of the C API. Chapter Eight builds on that to cover more advanced topics, such as storing times and dates, using SQLite from scripting languages, and utilizing some of the more advanced extensions. Chapters Nine and Ten cover writing your own custom SQL functions, extensions, and modules.

To complete the picture, the ten chapters are followed by several reference appendixes. These references cover all of the SQL commands, expressions, and built-in functions supported by SQLite, as well as documentation for the complete SQLite API.

SQLite Versions

The first edition of this book coves SQLite version 3.6.23.1. As this goes to press, work on SQLite version 3.7 is being finalized. SQLite 3.7 introduces a new transaction journal mode known as *Write Ahead Logging*, or WAL. In some environments, WAL can provide better concurrent transaction performance than the current rollback journal. This

performance comes at a cost, however. WAL has more restrictive operational requirements and requires more advanced support from the operating system.

Once WAL has been fully tested and released, look for an article on the O'Reilly website that covers this new feature and how to get the most out of it.

Email Lists

The SQLite project maintains three mailing lists. If you're trying to learn more about SQLite, or have any questions that are not addressed in this book or in the project documentation, these are often a good place to start.

sqlite-announce@sqlite.org
: This list is limited to announcements of new releases, critical bug alerts, and other significant events in the SQLite community. Traffic is extremely low, and most messages are posted by the SQLite development team.

sqlite-users@sqlite.org
: This is the main support list for SQLite. It covers a broad range of topics, including SQL questions, programming questions, and questions about how the library works. This list is moderately busy.

sqlite-dev@sqlite.org
: This list is for people working on the internal code of the SQLite library itself. If you have questions about how to use the published SQLite API, those questions belong on the *sqlite-users* list. Traffic on this list is fairly low.

You can find instructions on how to join these mailing lists on the SQLite website. Visit *http://www.sqlite.org/support.html* for more details.

The *sqlite-users@sqlite.org* email list can be quite helpful, but it is a moderately busy list. If you're only a casual user and don't wish to receive that much email, you can also access and search list messages through a web archive. Links to several different archives are available on the SQLite support page.

Example Code Download

The code examples found in this book are available for download from the O'Reilly website. You can find a link to the examples on the book's catalog page at *http://oreilly.com/catalog/9780596521196/*. The files include both the SQL examples and the C examples found in later chapters.

How We Got Here

Taking a book from an idea to a finished product involves a great many people. Although my name is on the cover, this could not have been possible without all of their help.

First, I would like to acknowledge the friendship and support of my primary editor, Mike Loukides. Thanks to some mutual friends, I first started doing technical reviews for Mike over eight years ago. Through the years, Mike gently encouraged me to take on my own project.

The first step on that path came nearly three years ago. I had downloaded a set of database exports from the Wikipedia project and was trying to devise a minimal database configuration that would (hopefully) cram nearly all the current data onto a small flash storage card. The end goal was to provide a local copy of the Wikipedia articles on an ebook reader I had. SQLite was a natural choice. At some point, frustrated with trying to understand the correct call sequence, I threw my hands up and exclaimed, "Someone should write a book about this!"—*Ding!*—The proverbial light bulb went off, and many, many (many...) late nights later, here we are.

Behind Mike stands the whole staff of O'Reilly Media. Everyone I interacted with did their best to help me out, calm me down, and fix my problems—sometimes all at once. The production staff understands how to make life easy for the author, so that we can focus on writing and leave the details to someone else.

I would like to thank D. Richard Hipp, the creator and lead maintainer of SQLite. In addition to coordinating the continued development of SQLite and providing us all with a high-quality software product, he was also gracious enough to answer numerous questions, as well as review a final draft of the manuscript. Some tricky spots went through several revisions, and he was always quick to review things and get back to me with additional comments.

A technical review was also done by Jon W. Marks. Jon is an old personal and professional friend with enterprise-class database experience. He has had the opportunity to mentor several experienced developers as they made their first journey into the relational database world. Jon provided very insightful feedback, and was able to pinpoint areas that are often difficult for beginners to grasp.

My final two technical reviewers were Jordan Hawker and Erin Moy. Although they are knowledgeable developers, they were relatively new to relational databases. As they went through the learning process, they kept me honest when I started to make too many assumptions, and kept me on track when I started to skip ahead too quickly.

I also owe a thank-you to Mike Kulas and all my coworkers at Volition, Inc. In addition to helping me find the right balance between my professional work and the book work, Mike helped me navigate our company's intellectual property policies, making sure everything was on the straight and narrow. Numerous coworkers also deserve a thank-you for reviewing small sections, looking at code, asking lots of good questions, and otherwise putting up with me venting about not having enough time in the day.

A tip of the hat goes out to the crew at the Aroma Café in downtown Champaign, Illinois. They're just a few blocks down from my workplace, and a significant portion of this book was written at their coffee shop. Many thanks to Michael and his staff, including Kim, Sara, Nichole, and Jerry, for always having a hot and creamy mocha ready.

Finally, I owe a tremendous debt to my wife, Debbie Fligor, and our two sons. They were always willing to make time for me to write and showed enormous amounts of patience and understanding. They all gave more than I had any right to ask, and this accomplishment is as much theirs as it is mine.

Conventions Used in This Book

The following typographical conventions are used in this book:

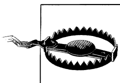
Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Using SQLite* by Jay A. Kreibich. Copyright 2010 O'Reilly Media, Inc., 978-0-596-52118-9."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that lets you easily search over 7,500 technology and creative reference books and videos to find the answers you need quickly.

With a subscription, you can read any page and watch any video from our library online. Read books on your cell phone and mobile devices. Access new titles before they are available for print, and get exclusive access to manuscripts in development and post feedback for the authors. Copy and paste code samples, organize your favorites, download chapters, bookmark key sections, create notes, print out pages, and benefit from tons of other time-saving features.

O'Reilly Media has uploaded this book to the Safari Books Online service. To have full digital access to this book and others on similar topics from O'Reilly and other publishers, sign up for free at *http://my.safaribooksonline.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

> O'Reilly Media, Inc.
> 1005 Gravenstein Highway North
> Sebastopol, CA 95472
> 800-998-9938 (in the United States or Canada)
> 707-829-0515 (international or local)
> 707 829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://oreilly.com/catalog/9780596521196/

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our web site at:

http://www.oreilly.com

CHAPTER 1

What Is SQLite?

In the simplest terms, SQLite is a public-domain software package that provides a *relational database management system*, or RDBMS. Relational database systems are used to store user-defined records in large tables. In addition to data storage and management, a database engine can process complex query commands that combine data from multiple tables to generate reports and data summaries. Other popular RDBMS products include Oracle Database, IBM's DB2, and Microsoft's SQL Server on the commercial side, with MySQL and PostgreSQL being popular open source products.

The "Lite" in SQLite does not refer to its capabilities. Rather, SQLite is lightweight when it comes to setup complexity, administrative overhead, and resource usage. SQLite is defined by the following features:

Serverless
: SQLite does not require a separate server process or system to operate. The SQLite library accesses its storage files directly.

Zero Configuration
: No server means no setup. Creating an SQLite database instance is as easy as opening a file.

Cross-Platform
: The entire database instance resides in a single cross-platform file, requiring no administration.

Self-Contained
: A single library contains the entire database system, which integrates directly into a host application.

Small Runtime Footprint
: The default build is less than a megabyte of code and requires only a few megabytes of memory. With some adjustments, both the library size and memory use can be significantly reduced.

Transactional
: SQLite transactions are fully ACID-compliant, allowing safe access from multiple processes or threads.

Full-Featured
: SQLite supports most of the query language features found in the SQL92 (SQL2) standard.

Highly Reliable
: The SQLite development team takes code testing and verification very seriously.

Overall, SQLite provides a very functional and flexible relational database environment that consumes minimal resources and creates minimal hassle for developers and users.

Self-Contained, No Server Required

Unlike most RDBMS products, SQLite does not have a client/server architecture. Most large-scale database systems have a large server package that makes up the database engine. The database server often consists of multiple processes that work in concert to manage client connections, file I/O, caches, query optimization, and query processing. A database instance typically consists of a large number of files organized into one or more directory trees on the server filesystem. In order to access the database, all of the files must be present and correct. This can make it somewhat difficult to move or reliably back up a database instance.

All of these components require resources and support from the host computer. Best practices also dictate that the host system be configured with dedicated service-user accounts, startup scripts, and dedicated storage, making the database server a very intrusive piece of software. For this reason, and for performance concerns, it is customary to dedicate a host computer system solely for the database server software.

To access the database, client software libraries are typically provided by the database vendor. These libraries must be integrated into any client application that wishes to access the database server. These client libraries provide APIs to find and connect to the database server, as well as set up and execute database queries and commands. Figure 1-1 shows how everything fits together in a typical client/server RDBMS.

In contrast, SQLite has no separate server. The entire database engine is integrated into whatever application needs to access a database. The only shared resource among applications is the single database file as it sits on disk. If you need to move or back up the database, you can simply copy the file. Figure 1-2 shows the SQLite infrastructure.

By eliminating the server, a significant amount of complexity is removed. This simplifies the software components and nearly eliminates the need for advanced operating system support. Unlike a traditional RDBMS server that requires advanced multitasking and high-performance inter-process communication, SQLite requires little more than the ability to read and write to some type of storage.

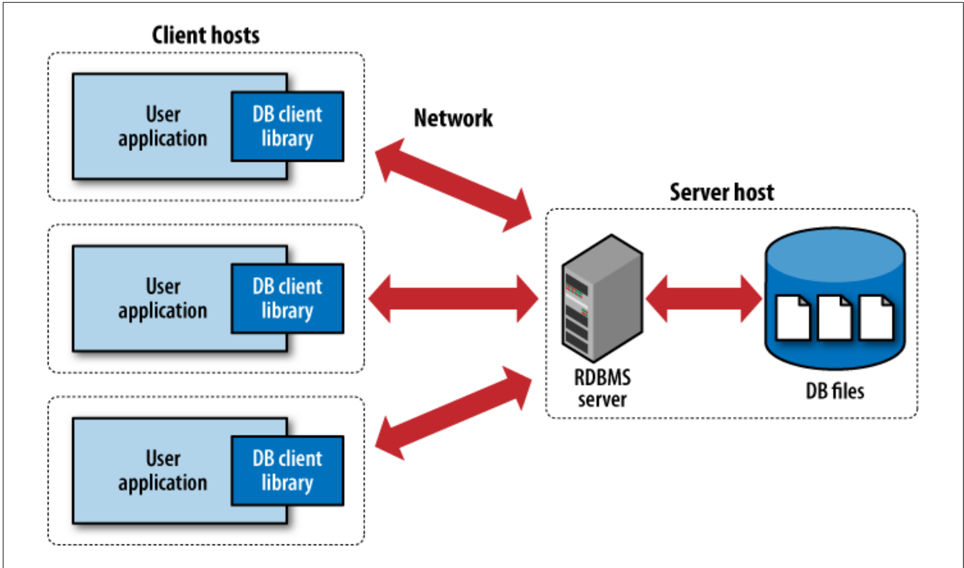


Figure 1-1. Traditional RDBMS client/server architecture that utilizes a client library.

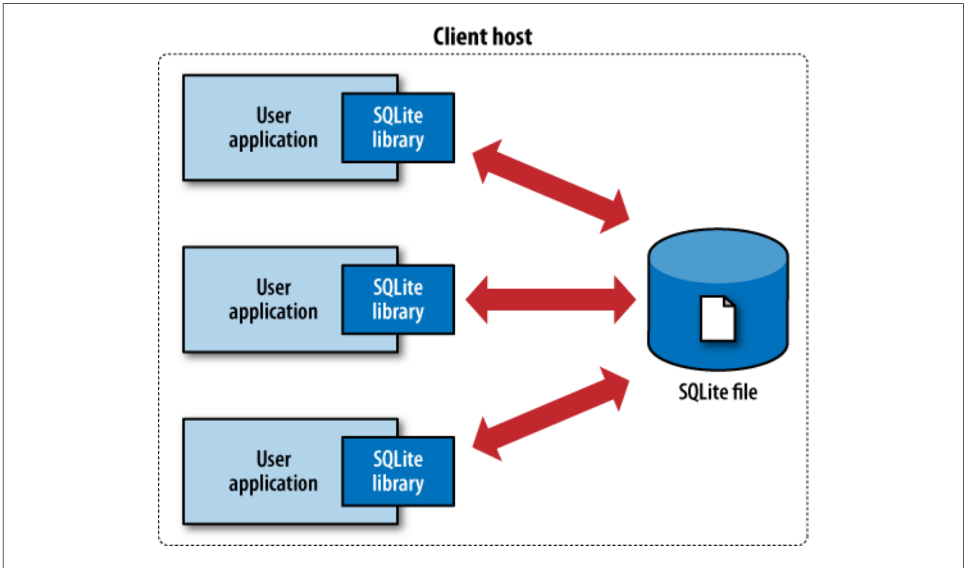
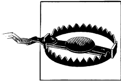


Figure 1-2. The SQLite server-less architecture.

This simplicity makes it fairly straightforward to port SQLite to just about any environment, including mobile phones, handheld media players, game consoles, and other devices, where no traditional database system could ever dare to venture.



Although SQLite does not use a traditional client/server architecture, it is common to speak of applications being "SQLite clients." This terminology is often used to describe independent applications that simultaneously access a shared SQLite database file, and is not meant to imply that there is a separate server.

SQLite is designed to be integrated directly into an executable. This eliminates the need for an external library and simplifies distribution and installation. Removing external dependencies also removes most versioning issues. If the SQLite code is built right into your application, you never have to worry about linking to the correct version of a client library, or that the client library is version-compatible with the database server.

Eliminating the server imposes some restrictions, however. SQLite is designed to address localized storage needs, such as a web server accessing a local database. This means it isn't well suited for situations where multiple client machines need to access a centralized database. That situation is more representative of a client/server architecture, and is better serviced by a database system that uses the same architecture.

Single File Database

SQLite packages the entire database into a single file. That single file contains the database layout as well as the actual data held in all the different tables and indexes. The file format is cross-platform and can be accessed on any machine, regardless of native byte order or word size.

Having the whole database in a single file makes it trivial to create, copy, or back up the on-disk database image. Whole databases can be emailed to colleagues, posted to a web forum, or checked into a revision control system. Entire databases can be moved, modified, and shared with the same ease as a word-processing document or spreadsheet file. There is no chance of a database becoming corrupt or unavailable because one of a dozen files was accidentally moved or renamed.

Perhaps most importantly, computer users have grown to expect that a document, project, or other "unit of application data" is stored as a single file. Having the whole database in a single file allows applications to use database instances as documents, data stores, or preference data, without contradicting customer expectations.

Zero Configuration

From an end-user standpoint, SQLite requires nothing to install, nothing to configure, and nothing to worry about. While there are a fair number of tuning parameters available to developers, these are normally hidden from the end-user. By eliminating the server and merging the database engine directly into your application, your customers never need to know they're using a database. It is quite practical to design an application

so that selecting a file is the only customer interaction—an action they are already comfortable doing.

Embedded Device Support

SQLite's small code size and conservative resource use makes it well suited for embedded systems running limited operating systems. The ANSI C source code tends toward an older, more conservative style that should be accepted by even the most eccentric embedded processor compiler. Using the default configuration, the compiled SQLite library is less than 700 KB on most platforms, and requires less than 4 MB of memory to operate. By omitting the more advanced features, the library can often be trimmed to 300 KB or less. With minor configuration changes, the library can be made to function on less than 256 KB of memory, making its total footprint not much more than half a megabyte, plus data storage.

SQLite expects only minimal support from its host environment and is written in a very modular way. The internal memory allocator can be easily modified or swapped out, while all file and storage access is done through a *Virtual File System* (VFS) interface that can be modified to meet the needs and requirements of different platforms. In general, SQLite can be made to run on almost anything with a 32-bit processor.

Unique Features

SQLite offers several features not found in many other database systems. The most notable difference is that SQLite uses a dynamic-type system for tables. The SQLite engine will allow you to put any value into nearly any column, regardless of type. This is a major departure from traditional database systems, which tend to be statically typed. In many ways, the dynamic-type system in SQLite is similar to those found in popular scripting languages, which often have a single scalar type that can accept anything from integers to strings. In my own experience, the dynamic-type system has solved many more problems than it has caused.

Another useful feature is the ability to manipulate more than one database at a time. SQLite allows a single database connection to associate itself with multiple database files simultaneously. This allows SQLite to process SQL statements that bridge across multiple databases. This makes it trivial to join tables from different databases with a single query, or bulk copy data with a single command.

SQLite also has the ability to create fully in-memory databases. These are essentially database "files" that have no backing store and spend their entire lifetime within the file cache. While in-memory databases lack durability and do not provide full transaction support, they are very fast (assuming you have enough RAM), and are a great place to store temporary tables and other transient data.

There are a number of other features that make SQLite extremely flexible. Many of these, like virtual tables, are based off similar features found in other products, but with their own unique twist. These features and extensions provide a number of powerful tools to adapt SQLite to your own particular problem or situation.

Compatible License

SQLite, and the SQLite code, have no user license. It is not covered by the *GNU General Public License* (GPL) or any of the similar open source/free-source licenses. Rather, the SQLite development team has chosen to place the SQLite source code in the public domain. This means that they have explicitly and deliberately relinquished any claim they have to copyright or ownership of the code or derived products.

In short, this basically means you can do whatever you want with the SQLite source code, short of claiming to own it. The code and compiled libraries can be used in any way, modified in any way, redistributed in any way, and sold in any way. There are no restrictions, and no requirements or obligations to make third-party changes or modifications available to the project or the public.

The SQLite team takes this decision very seriously. Great care is taken to avoid any potential software patents or patented algorithms. All contributions to the SQLite source require a formal copyright release. Commercial contributions also require signed affidavits stating that the authors (and, if applicable, their employers) release their work into the public domain.

All of this effort is taken to ensure that integrating SQLite into your product carries along minimal legal baggage or liability, making it a viable option for almost any development effort.

Highly Reliable

The purpose of a database is to keep your data safe and organized. The SQLite development team is aware that nobody will use a database product that has a reputation for being buggy or unreliable. To maintain a high level of reliability, the core SQLite library is aggressively tested before each release.

In full, the standard SQLite test suites consists of over 10 million unit tests and query tests. The "soak test," done prior to each release, consists of over 2.5 billion tests. The suite provides 100% statement coverage and 100% branch coverage, including edge-case errors, such as out-of-memory and out-of-storage conditions. The test suite is designed to push the system to its specified limits and beyond, providing extensive coverage of both code and operating parameters.

This high level of testing keeps the SQLite bug count relatively low. No software is perfect, but bugs that contribute to actual data-loss or database corruption are fairly rare. Most bugs that escape are performance related, where the database will do the right thing, but in the wrong way, leading to longer run-times.

Strong testing also keeps backwards compatibility extremely solid. The SQLite team takes backwards compatibility very seriously. File formats, SQL syntax, and programming APIs and behaviors have an extremely strong history of backwards compatibility. Updating to a new version of SQLite rarely causes compatibility problems.

In addition to keeping the core library reliable, the extensive testing also frees the SQLite developers to be more experimental. Whole subsystems of the SQLite code can be (and have been) ripped out and updated or replaced, with little concern about compatibility or functional differences—as long as all the tests pass. This allows the team to make significant changes with relatively little risk, constantly pushing the product and performance forward.

Like so many other aspects of the SQLite design, fewer bugs means fewer problems and less to worry about. As much as any complex piece of software can, it just works.

CHAPTER 2

Uses of SQLite

SQLite is remarkably flexible in both where it can be run and how it can be used. This chapter will take a brief look at some of the roles SQLite is designed to fill. Some of these roles are similar to those taken by traditional client/server RDBMS products. Other roles take advantage of SQLite's size and ease of use, offering solutions you might not consider with a full client/server database.

Database Junior

Years of experience has taught developers that large client/server RDBMS platforms are powerful tools for safely storing, organizing, and manipulating data. Unfortunately, most large RDBMS products are resource-intensive and require a lot of upkeep. This boosts their performance and capacity, but it also limits how and where they can be practically deployed.

SQLite is designed to fill in those gaps, providing the same powerful and familiar tools for safely storing, organizing, and manipulating data in smaller, more resource constrained environments. SQLite is designed to complement, rather than replace, larger RDBMS platforms in situations where simplicity and ease of use are more important than capacity and concurrency.

This complimentary role enables applications and tools to embrace relational data management (and the years of experience that come with it), even if they're running on smaller platforms without administrative oversight. Developers may laugh at the idea of installing MySQL on a desktop system (or mobile phone!) just to support an address book application, but with SQLite this not only becomes possible, it becomes entirely practical.

Application Files

Modern desktop applications typically deal with a significant number of files. Most applications and utilities have one or more preference files. There may also be system-wide and per-user configuration files, caches, and other data that must be tracked and stored. Document-based applications also need to store and access the actual document files.

Using the SQLite library as an abstract storage layer has many advantages. A fair amount of application metadata, such as caches, state data, and configuration data, fit well with the relational data model. This makes it relatively easy to create an appropriate database design that maps cleanly and easily into an application's internal data structures.

In many cases, SQLite can also work well as a document file format. Rather than creating a custom document format, an application can simply use individual database instances to represent working documents. SQLite supports many standard datatypes, including Unicode text, as well as arbitrary binary data fields that can store images or other raw data.

Even if an application does not have particularly strong relational requirements, there are still significant advantages to using the SQLite library as a storage container. The SQLite library provides incremental updates that make it quick and easy to save small changes. The transaction system protects all file I/O against process termination and power disruption, nearly eliminating the possibility of file corruption. SQLite even provides its own file caching layer, so that very large files can be opened and processed in a limited memory footprint, without any additional work on the part of the application.

SQLite database files are cross-platform, allowing easy migration. File contents can be easily and safely shared with other applications without worrying about detailed file format specifications. The common file format also makes it easy for automated scripts or troubleshooting utilities to access the files. Multiple applications can even access the same file simultaneously, and the library will transparently take care of all required file locking and cache synchronization.

The use of SQLite can also make debugging and troubleshooting much easier, as files can be inspected and manipulated with standard database tools. You can even use standard tools to inspect and modify a database file as your application is using it. Similarly, test files can be programmatically generated outside the application, which is useful for automatic testing suites.

Using an entire database instance as a document container may sound a bit unusual, but it is worth considering. The advantages are significant and should help a developer stay focused on the core of their application, rather than worrying about file formats, caching, or data synchronization.

Application Cache

SQLite is capable of creating databases that are held entirely in memory. This is extremely useful for creating small, temporary databases that require no permanent storage.

In-memory databases are often used to cache results pulled from a more traditional RDBMS server. An application may pull a subset of data from the remote database, place it into a temporary database, and then process multiple detailed searches and refinements against the local copy. This is particularly useful when processing type-ahead suggestions, or any other interactive element that requires very quick response times.

Temporary databases can also be used to index and store nearly any type of inter-linked, cross-referenced data. Rather than designing a set of complex runtime data structures which might include multiple hash tables, trees, and cross-referenced pointers, the developer can simply design an appropriate database schema and load the data into the database.

While it might seem odd to execute SQL statements in order to extract data from an in-memory data structure, it is surprisingly efficient and can reduce development time and complexity. A database also provides an upgrade path, making it trivial to grow the data beyond the available memory or persist the data across application runs, simply by migrating to an on-disk database.

Archives and Data Stores

SQLite makes it very easy to package complex data sets into a single, easy-to-access, fully cross-platform file. Having all the data in a single file makes it much easier to distribute or download large, multi-table data stores, such as large dictionaries or geo-location references.

Unlike many RDBMS products, the SQLite library is able to access read-only database files. This allows data stores to be read directly from an optical disc or other read-only filesystem. This is especially useful for systems with limited hard drive space, such as video game consoles.

Client/Server Stand-in

SQLite works well as a "stand-in" database for those situations when a more robust RDBMS would normally be the right choice, were it available. SQLite can be especially useful for the demonstration and evaluation of applications and tools that normally depend on a database.

Consider a data analysis product that is designed to pull data from a relational database to generate reports and graphs. It can be difficult to offer downloads and evaluation

copies of such software. Even if a download is available, the software must be configured and authorized to connect to a database that contains applicable data. This presents a significant barrier for a potential customer.

Now consider an evaluation download that includes support for a bundled SQLite demonstration database. By simply downloading and running the software, customers can interact and experiment with the sample database. This makes the barrier of entry significantly lower, allowing a customer to go from downloading to running data in just a few seconds.

Similar concerns apply to traditional sales and marketing demonstrations. Reliable network connectivity is often unavailable when doing on-site demonstrations to potential clients, so it is standard practice to run a local database server for demonstration purposes. Running a local database server consumes significant resources and adds administrative overhead. Database licenses may also be a concern.

The use of SQLite eliminates these issues. The database becomes a background piece, allowing the demonstration to focus on the product. There are no database administration concerns. The simple file format also makes it easy to prepare customer-specific data sets or show off product features that significantly modify the database. All this can be done by simply making a copy of the database file before proceeding.

Beyond evaluations and demonstrations, SQLite support can be used to promote a "lite" or "personal edition" of a larger product. Adding an entry-level product that is more suitable and cost-effective for smaller installations can open up a significant number of new customers by providing a low-cost, no-fuss introduction to the product line.

SQLite support can even help with development and testing. SQLite databases are small and compact, allowing them to be attached to bug reports. They also provide an easy way to test a wide variety of situations, allowing a product to be tested against hundreds, if not thousands, of unique database instances. Even if a customer never sees an SQLite database, the integration time may easily pay for itself with improved testing and debugging capabilities.

Teaching Tool

For the student looking to learn SQL and the basics of the relational model, SQLite provides an extremely accessible environment that is easy to set up, easy to use, and easy to share. SQLite offers a full-fledged relational system that supports nearly all of the core SQL language, yet requires no server setup, no administration, and no overhead. This allows students to focus on learning SQL and data manipulation without getting bogged down by server configuration and database maintenance.

Given its compact size, it is simple to place a Windows, Mac OS X, and Linux version of the command-line tools, along with several databases, onto a small flash drive. With no installation process and fully cross-platform database files, this provides an "on the go" teaching environment that will work with nearly any computer.

The "database in a file" architecture makes it easy for students to share their work. Whole database instances can be attached to an email or posted to a discussion forum. The single-file format also makes it trivial to back up work in progress, allowing students to experiment and explore different solutions without concern over losing data.

Generic SQL Engine

SQLite virtual tables allow a developer to define the contents of a table through code. By defining a set of callback functions that fetch and return rows and columns, a developer can create a link between the SQLite data processing engine and any data source. This allows SQLite to run queries against the data source without importing the data into a standard table.

Virtual tables are an extremely useful way to generate reports or allow ad hoc queries against logs or any tabular data set. Rather than writing a set of custom search or reporting tools, the data can simply be exposed to the SQLite engine. This allows reports and queries to be expressed in SQL, a language that many developers are already familiar with using. It also enables the use of generic database visualization tools and report generators.

Chapter 10 shows how to build a virtual table module that provides direct access to live web server logs.

Not the Best Choice

Although SQLite has proven itself extremely flexible, there are some roles that are outside of its design goals. While SQLite may be able to perform in these areas, it might not be the best fit. If you find yourself with any of these requirements, it may be more practical to consider a more traditional client/server RDBMS product.

High Transaction Rates
: SQLite is able to support moderate transaction rates, but it is not designed to support the level of concurrent access provided by many client/server RDBMS products. Many server systems are able to provide table-level or row-level locking, allowing multiple transactions to be processed in parallel without the risk of data loss.

The concurrency protection offered by SQLite depends on file locks to protect against data loss. This model allows multiple database connections to access a database at the same time, but the whole database file must be locked in an exclusive mode to make any changes. As a result, write transactions are serialized across all database connections, limiting the overall transaction rate.

Depending on the size and complexity of your updates, SQLite might be able to handle a few hundred transactions per minute from different processes or threads. If, however, you start to see performance problems, or expect higher transaction rates, a client/server system is likely to provide better transaction performance.

Extremely Large Datasets

It is not unusual to find SQLite databases that approach a dozen gigabytes or more, but there are some practical limits to the amount of data that can (or should) be stuffed into an SQLite database. Because SQLite puts everything into a single file (and thus, a single filesystem), very large data sets can stress the capability of the operating system or filesystem design. Although most modern filesystems are capable of handling files that are a terabyte or larger, that doesn't always mean they're very good at it. Many filesystems see a significant drop in performance for random access patterns if the file starts to get into multiple gigabyte ranges.

If you need to store and process several gigabytes or more of data, it might be wise to consider a more performance-oriented product.

Access Control

An SQLite database has no authentication or authorization data. Instead, SQLite depends on filesystem permissions to control access to the raw database file. This essentially limits access to one of three states: complete read/write access, read-only access, or no access at all. Write access is absolute, and allows both data modification and the ability to alter the structure of the database itself.

While the SQLite API provides a basic application-layer authorization mechanism, it is trivial to circumvent if the user has direct access to the database file. Overall, this makes SQLite unsuitable for sensitive data stores that require per-user access control.

Client/Server

SQLite is specifically designed without a network component, and is best used as a local resource. There is no native support for providing access to multiple computers over a network, making it a poor choice as a client/server database system.

Having multiple computers access an SQLite file through a shared directory is also problematic. Most networked filesystems have poor file-locking facilities. Without the ability to properly lock the file and keep updates synchronized, the database file can easily become corrupt.

This isn't to say that client/server systems can't utilize SQLite. For example, many web servers utilize SQLite. This works because all of the web server processes are running on the same machine and are all accessing the database file from local storage.

Replication

SQLite has no internal support for database replication or redundancy. Simple replication can be achieved by copying the database file, but this must be done when nothing is attempting to modify the database.

Replication systems can be built on top of the basic database API, but such systems tend to be somewhat fragile. Overall, if you're looking for real-time replication—especially at a transaction-safe level—you'll need to look at a more complex RDBMS platform.

Most of these requirements get into a realm where complexity and administrative overhead is traded for capacity and performance. This makes sense for a large client/server RDBMS platform, but it is somewhat at odds with the SQLite design goals of staying simple and maintenance free. To keep frustration to a minimum, use the right tool for the job.

Big Name Users

The SQLite website states that, "*SQLite is the most widely deployed SQL database engine in the world.*" This is a pretty bold claim, especially considering that when most people think of relational database platforms, they usually think of names like Oracle, SQL Server, and MySQL.

It is also a claim that is difficult to support with exact numbers. Because there are no license agreements or disclosure requirements, it is hard to guess just how many SQLite databases are out there. Nobody, including the SQLite development team, is fully aware of who is using SQLite, and for what purposes.

Regardless, the list of known SQLite users adds up to an impressive list. The Firefox web browser and the Thunderbird email client both use several SQLite databases to store cookies, history, preferences, and other account data. Many products from Skype, Adobe, and McAfee also utilize the SQLite engine. The SQLite library is also integrated into a number of popular scripting languages, including PHP and Python.

Apple, Inc., has heavily embraced SQLite, meaning that every iPhone, iPod touch, and iPad, plus every copy of iTunes, and many other Macintosh applications, all ship with several SQLite databases. The Symbian, Android, BlackBerry, and Palm webOS environments all provide native SQLite support, while WinCE has third-party support. Chances are, if you have a smartphone, it has a number of SQLite databases stored on it.

All of this adds up to millions, if not billions, of SQLite databases in the wild. No doubt that most of these databases only contain a few hundred kilobytes of data, but these low-profile environments are exactly where SQLite is designed to thrive.

Large client/server RDBMS platforms have shown thousands of developers the power of relational data management systems. SQLite has brought that power out of the server room to the desktops and mobile devices of the world.

CHAPTER 3
Building and Installing SQLite

This chapter is about building SQLite. We'll cover how to build and install the SQLite distribution on Linux, Mac OS X, and Windows. The SQLite code base supports all of these operating systems natively, and precompiled libraries and executables for all three environments are available from the SQLite website. All downloads, including source and precompiled binaries, can be found on the SQLite download webpage (*http://www.sqlite.org/download.html*).

SQLite Products

The SQLite project consists of four major products:

SQLite core
: The SQLite core contains the actual database engine and public API. The core can be built into a static or dynamic library, or it can be built in directly to an application.

sqlite3 command-line tool
: The `sqlite3` application is a command-line tool that is built on top of the SQLite core. It allows a developer to issue interactive SQL commands to the SQLite core. It is extremely useful for developing and debugging queries.

Tcl extension
: SQLite has a strong history with the Tcl language. This library is essentially a copy of the SQLite core with the Tcl bindings tacked on. When compiled into a library, this code exposes the SQLite interfaces to the Tcl language through the *Tcl Extension Architecture* (TEA). Outside of the native C API, these Tcl bindings are the only official programming interface supported directly by the SQLite team.

SQLite analyzer tool

The SQLite analyzer is used to analyze database files. It displays statistics about the database file size, fragmentation, available free space, and other data points. It is most useful for debugging performance issues related to the physical layout of the database file. It can also be used to determine if it is appropriate to `VACUUM` (repack and defragment) the database or not. The SQLite website provides precompiled `sqlite3_analyzer` executables for most desktop platforms. The source for the analyzer is only available through the development source distribution.

Most developers will be primarily interested in the first two products: the SQLite core and the `sqlite3` command-line tool. The rest of the chapter will focus on these two products. The build process for the Tcl extension is identical to building the SQLite core as a dynamic library. The analyzer tool is normally not built, but simply downloaded. If you want to build your own copy from scratch, you need a full development tree to do so.

Precompiled Distributions

The SQLite download page includes precompiled, standalone versions of the `sqlite3` command-line tool for Linux, Mac OS X, and Windows. If you want to get started experimenting with SQLite, you can simply download the command-line tool, unpack it, run it, and start issuing SQL commands. You may not even have to download it first—Mac OS X and most Linux distributions include a copy of the `sqlite3` utility as part of the operating system. The SQLite download page also includes precompiled, standalone versions of the `sqlite3_analyzer` for all three operating systems.

Precompiled dynamic libraries of the SQLite core and the Tcl extension are also available for Linux and Windows. The Linux files are distributed as shared objects (*.so* files), while the Windows downloads contain DLL files. No precompiled libraries are available for Mac OS X. The libraries are only required if you are writing your own application, but do not wish to compile the SQLite core directly into your application.

Documentation Distribution

The SQLite download page includes a documentation distribution. The *sqlite_docs_3_x_x.zip* file contains most of the static content from the SQLite website. The documentation online at the SQLite website is not versioned and always reflects the API and SQL syntax for the most recent version of SQLite. If you don't plan on continuously upgrading your SQLite distribution, it is useful to grab a copy of the documentation that goes with the version of SQLite you are using.

Source Distributions

Most open source projects provide a single download that allows you to configure, build, and install the software with just a handful of commands. SQLite works a bit differently. Because the most common way to use SQLite is to integrate the core source directly into a host application, the source distributions are designed to make integration as simple as possible. Most of the source distributions contain only source code and provide minimal (if any) configuration or build support files. This makes it simpler to integrate SQLite into a host application, but if you want to build a library or `sqlite3` application, you will often need to do that by hand. As we'll see, that's fairly easy.

The Amalgamation

The official code distribution is known as the *amalgamation*. The amalgamation is a single C source file that contains the entire SQLite core. It is created by assembling the individual development files into a single C source file that is almost 4 megabytes in size and over 100,000 lines long. The amalgamation, along with its corresponding header file, is all that is needed to integrate SQLite into your application.

The amalgamation has two main advantages. First, with everything in one file, it is extremely easy to integrate SQLite into a host application. Many projects simply copy the amalgamation files into their own source directories. It is also possible to compile the SQLite core into a library and simply link the library into your application.

Second, the amalgamation also helps improve performance. Many compiler optimizations are limited to a single translation unit. In C, that's a single source file. By putting the whole library into a single file, a good optimizer can process the whole package at once. Compared to compiling the individual source files, some platforms see a 5% or better performance boost just by using the amalgamation.

The only disadvantage of using the amalgamation is size. Some debuggers have issues with files more than 65,535 lines long. Things typically run correctly, but it can be difficult to set breakpoints or look at stack traces. Compiling a source file over 100,000 lines long also takes a fair number of resources. While this is no problem for most desktop systems, it may push the limits of any compilers running on limited platforms.

Source Files

When working with the amalgamation, there are four important source files:

sqlite3.c
: The amalgamation source file, which includes the entire SQLite core, plus common extensions.

sqlite3.h
: The amalgamation header file, which exposes the core API.

sqlite3ext.h
: The extension header file, which is used to build SQLite extensions.

shell.c
: The `sqlite3` application source, which provides an interactive command-line shell.

The first two, *sqlite3.c* and *sqlite3.h*, are all that is needed to integrate SQLite into most applications. The *sqlite3ext.h* file is used to build extensions and modules. Building extensions is covered in "SQLite Extensions" on page 204. The *shell.c* file contains the source code for the `sqlite3` command-line shell. All of these files can be built on Linux, Mac OS X, or Windows, without any additional configuration files.

Source Downloads

The SQLite website offers five source distribution packages. Most people will be interested in one of the first two files.

sqlite-amalgamation-3_x_x.zip
: The Windows amalgamation distribution.

sqlite-amalgamation-3.x.x.tar.gz
: The Unix amalgamation distribution.

sqlite-3_x_x-tea.tar.gz
: The Tcl extension distribution.

sqlite-3.x.x.tar.gz
: The Unix source tree distribution. This is unsupported and the build files are unmaintained.

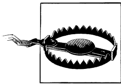
sqlite-source-3_x_x.zip
: The Windows source distribution. This is unsupported.

The Windows amalgamation file consists of the four main files, plus a *.def* file to build a DLL. No makefile, project, or solution files are included.

The Unix amalgamation file, which works on Linux, Mac OS X, and many other flavors of Unix, contains the four main files plus an `sqlite3` manual page. The Unix distribution also contains a basic *configure* script, along with other autoconf files, scripts, and makefiles. The autoconf files should also work under the Minimalist GNU for Windows (MinGW) environment (*http://www.mingw.org/*).

The Tcl extension distribution is a specialized version of the amalgamation. It is only of interest to those working in the Tcl language. See the included documentation for more details.

The Unix source tree is an unsupported legacy distribution. This is what the standard distribution looked like before the amalgamation became the officially supported distribution. It is made available for those that have older build environments or development branches that utilize the old distribution format. This distribution also includes a number of README files that are unavailable elsewhere.



Although the source files are kept up to date, the configuration scripts and makefiles included in the Unix source tree distribution are no longer maintained and do not work properly on most platforms. Unless you have some significant need to use the source tree distribution, you should use one of the amalgamation distributions instead.

The Windows source distribution is essentially a *.zip* file of the source directory from the source tree distribution, minus some test files. It is strictly source files and header files, and contains no build scripts, makefiles, or project files.

Building

There are a number of different ways to build SQLite, depending on what you're trying to build and where you would like it installed. If you are trying to integrate the SQLite core into a host application, the easiest way to do that is to simply copy *sqlite3.c* and *sqlite3.h* into your application's source directory. If you're using an IDE, the *sqlite3.c* file can simply be added to your application's project file and configured with the proper search paths and build directives. If you want to build a custom version of the SQLite library or `sqlite3` utility, it is also easy to do that by hand.

All of the SQLite source is written in C. It cannot be compiled by a C++ compiler. If you're getting errors related to structure definitions, chances are you're using a C++ compiler. Make sure you use a vanilla C compiler.

Configure

If you're using the Unix amalgamation distribution, you can build and install SQLite using the standard *configure* script. After downloading the distribution, it is fairly easy to unpack, configure, and build the source:

```
$ tar xzf sqlite-amalgamation-3.x.x.tar.gz
$ cd sqlite-3.x.x
$ ./configure
  [...]
$ make
```

By default, this will build the SQLite core into both static and dynamic libraries. It will also build the `sqlite3` utility. These will be built with many of the extra features (such as full text search and R*Tree support) enabled. Once this finishes, the command `make install` will install these files, along with the header files and `sqlite3` manual page. By

default, everything is installed into `/usr/local`, although this can be changed by giving a `--prefix=/path/to/install` option to `configure`. Issue the command `configure --help` for information on other build options.

Manually

Because the main SQLite amalgamation consists of only two source files and two header files, it is extremely simple to build by hand. For example, to build the `sqlite3` shell on Linux, or most other Unix systems:

```
$ cc -o sqlite3 shell.c sqlite3.c -ldl -lpthread
```

The additional libraries are needed to support dynamic linking and threads. Mac OS X includes those libraries in the standard system group, so no additional libraries are required when building for Mac OS X:

```
$ cc -o sqlite3 shell.c sqlite3.c
```

The commands are very similar on Windows, using the Visual Studio C compiler from the command-line:

```
> cl /Fesqlite3 shell.c sqlite3.c
```

This will build both the SQLite core and the shell into one application. That means the resulting `sqlite3` executable will not require an installed library in order to operate.

If you want to build things with one of the optional modules installed, you need to define the appropriate compiler directives. This shows how to build things on Unix with the FTS3 (full text search) extension enabled:

```
$ cc -DSQLITE_ENABLE_FTS3 -o sqlite3 shell.c sqlite3.c -ldl -lpthread
```

Or, on Windows:

```
> cl /Fesqlite3 /DSQLITE_ENABLE_FTS3 shell.c sqlite3.c
```

Building the SQLite core into a dynamic library is a bit more complex. We need to build the object file, then build the library using that object file. If you've already built the `sqlite3` utility, and have an *sqlite3.o* (or *.obj*) file, you can skip the first step. First, in Linux and most Unix systems:

```
$ cc -c sqlite3.c
$ ld -shared -o libsqlite3.so sqlite3.o
```

Some versions of Linux may also require the `-fPIC` option when compiling.

Mac OS X uses a slightly different dynamic library format, so the command to build it is slightly different. It also needs the standard C library to be explicitly linked:

```
$ cc -c sqlite3.c
$ ld -dylib -o libsqlite3.dylib sqlite3.o -lc
```

And finally, building a Windows DLL (which requires the *sqlite3.def* file):

```
> cl /c sqlite3.c
> link /dll /out:sqlite3.dll /def:sqlite3.def sqlite3.obj
```

You may need to edit the *sqlite3.def* file to add or remove functions, depending on which compiler directives are used.

Build Customization

The SQLite core is aware of a great number of compiler directives. Appendix A covers all of these in detail. Many are used to alter the standard default values, or to adjust some of the maximum sizes and limits. Compiler directives are also used to enable or disable specific features and extensions. There are several dozen directives in all.

The default build, without any specific directives, will work well enough for a wide variety of applications. However, if your application requires one of the extensions, or has specific performance concerns, there may be some ways to tune the build. Many of the parameters can also be adjusted at runtime, so a recompile may not always be necessary, but it can make development more convenient.

Build and Installation Options

There are several different ways to build, integrate, and install SQLite. The design of the SQLite core lends itself to being compiled as a dynamic library. A single library can then be utilized by whatever application requires SQLite.

Building a shared library this way is one of the more straightforward ways to integrate and install SQLite, but it is often not the best approach. The SQLite project releases new versions rather frequently. While they take backward compatibility seriously, there are sometimes changes to the default configuration. There are also cases of applications becoming dependent on version-specific bugs or undefined (or undocumented) behaviors. There are also a large number of custom build options that SQLite supports. All these concerns can be difficult to create a system-wide build that is suitable for every application that uses SQLite.

This problem becomes worse as the number of applications utilizing SQLite continues to increase, making for more and more application-specific copies of SQLite. Even if an application (or suite of applications) has its own private copy of an SQLite library, there is still the possibility of incorrect linking and version incompatibilities.

To avoid these problems, the recommended way of using SQLite is to integrate the whole database engine directly into your application. This can be done by building a static library and then linking it in, or by simply building the amalgamation source directly into your application code. This method provides a truly custom build that is tightly bound to the application that uses it, eliminating any possibility of version or build incompatibilities.

About the only time it may be appropriate to use a dynamic library is when you're building against an existing system-installed (and system-maintained) library. This includes Mac OS X, many Linux distributions, as well as the majority of phone environments. In that case, you're depending on the operating system to keep a consistent build. This normally works for reasonably simple needs, but your application needs to be somewhat flexible. System libraries are often frozen with each major release, but chances are that sooner or later the system software (including the SQLite system libraries) will be upgraded. Your application may have to work across different versions of the system library if you need to support different versions of the operating system. For all these same reasons, it is ill-advised to manually replace or upgrade the system copy of SQLite.

If you do decide to use your own private library, take great care when linking. It is all too easy to accidentally link against a system library, rather than your private copy, if both are available.

Versioning problems, along with many other issues, can be completely avoided if the application simply contains its own copy of the SQLite core. The SQLite source distributions and the amalgamation make direct integration an easy path to take. Libraries have their place, but makes sure you understand the possible implications of having an external library. In specific, unless you control an entire device, never assume you're the only SQLite user. Try to keep your builds and installs clear of any system-wide library locations.

An sqlite3 Primer

Once you have some form of SQLite installed, the first step is normally to run `sqlite3` and play around. The `sqlite3` tool accepts SQL commands from an interactive prompt and passes those commands to the SQLite core for processing.

Even if you have no intention of distributing a copy of `sqlite3` with your application, it is extremely useful to have a copy around for testing and debugging queries. If your application uses a customized build of the SQLite core, you will likely want to build a copy of `sqlite3` using the same build parameters.

To get started, just run the SQLite command. If you provide a filename (such as *test.db*), sqlite3 will open (or create) that file. If no filename is given, `sqlite3` will automatically open an unnamed temporary database:

```
$ sqlite3 test.db
SQLite version 3.6.23.1
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

The `sqlite>` prompt means `sqlite3` is ready to accept commands. We can start with some basic expressions:

```
sqlite> SELECT 3 * 5, 10;
15|10
sqlite>
```

SQL commands can also be entered across multiple lines. If no terminating semicolon is found, the statement is assumed to continue. In that case, the prompt will change to `...>` to indicate `sqlite3` is waiting for more input:

```
sqlite> SELECT 1 + 2,
   ...> 6 + 3;
3|9
sqlite>
```

If you ever find yourself at the `...>` prompt unexpectedly, make sure you finished up the previous line with a semicolon.

In addition to processing SQL statements, there is a series of shell-specific commands. These are sometimes referred to as "dot-commands" because they start with a period. Dot-commands control the shell's output formatting, and also provide a number of utility features. For example, the `.read` command can be used to execute a file full of SQL commands.

Dot-commands must be given at the `sqlite>` prompt. They must be given on one line, and should not end in a semicolon. You cannot mix SQL statements and dot-commands.

Two of the more useful dot-commands (besides `.help`) are `.headers` and `.mode`. Both of these control some aspect of the database output. Turning headers on and setting the output mode to `column` will produce a table that most people find easier to read:

```
sqlite> SELECT 'abc' AS start, 'xyz' AS end;
abc|xyz
sqlite> .headers on
sqlite> .mode column
sqlite> SELECT 'abc' AS start, 'xyz' AS end;
start       end
----------  ----------
abc         xyz
sqlite>
```

Also helpful is the `.schema` command. This will list all of the DDL commands (`CREATE TABLE`, `CREATE INDEX`, etc.) used to define the database. For a more complete list of all the `sqlite3` command-line options and dot-commands, see Appendix A.

Summary

SQLite is designed to integrate into a wide variety of code bases on a broad range of platforms. This flexibility provides a great number of options, even for the most basic situations. While flexibility is usually a good thing, it can make for a lot of confusion when you're first trying to figure things out.

If you're just starting out, and all you need is a copy of the `sqlite3` shell, don't get too caught up in all the advanced build techniques. You can download one of the precompiled executables or build your own with the one-line commands provided in this chapter. That will get you started.

As your needs evolve, you may need a more specific build of `sqlite3`, or you may start to look at integrating the SQLite library into your own application. At that point you can try out different build techniques and see what best matches your needs and build environment.

While the amalgamation is a somewhat unusual form for source distribution, it has proven itself to be quite useful and well suited for integrating SQLite into larger projects with the minimal amount of fuss. It is also the only officially supported source distribution format. It works well for the majority of projects.

CHAPTER 4

The SQL Language

This chapter provides an overview of the *Structured Query Language*, or *SQL*. Although sometimes pronounced “sequel,” the official pronunciation is to name each letter as “ess-cue-ell.” The SQL language is the main means of interacting with nearly all modern relational database systems. SQL provides commands to configure the tables, indexes, and other data structures within the database. SQL commands are also used to insert, update, and delete data records, as well as query those records to look up specific data values.

All interaction with a relational database is done through the SQL language. This is true when interactively typing commands or when using the programming API. In all cases, data is stored, modified, and retrieved through SQL commands. Many times, people look through the list of API calls, looking for functions that provide direct program access to the table or index data structures. Functions of this sort do not exist. The API is structured around preparing and issuing SQL commands to the database engine. If you want to query a table or insert a value using the API, you must create and execute the proper SQL command. If you want to do relational database programming, you must know SQL.

Learning SQL

The goal of this chapter is to introduce you to all the major SQL commands and show some of the basic usage patterns. The first time you read through this chapter, don’t feel you need to absorb everything at once. Get an idea of what structures the database supports, and how they might be used, but don’t feel that you need to memorize the details of every last command.

For people just getting started, the most important commands are `CREATE TABLE`, `INSERT`, and `SELECT`. These will let you create a table, insert some data into the table, and then query the data and display it. Once you get comfortable with those commands, you can start to look at the others in more depth. Feel free to refer back to this chapter, or the command reference in Appendix C. The command reference provides detailed descriptions of each command, including some of the more advanced syntax that isn't covered in this chapter.

Always remember that SQL is a command language. It assumes you know what you're doing. If you're directly entering SQL commands through the `sqlite3` application, the program will not stop and ask for confirmation before processing dangerous or destructive commands. When entering commands by hand, it is always worth pausing and looking back at what you've typed before you hit return.

If you are already reasonably familiar with the SQL language, it should be safe to skim this chapter. Much of the information here is on the SQL language in general, but there is some information about the specific dialect of SQL that SQLite recognizes. Again, Appendix C provides a reference to the specific SQL syntax used by SQLite.

Brief Background

Although the first official SQL specification was published in 1986 by the American National Standards Institute (ANSI), the language traces its roots back to the early 1970s and the pioneering relational database work that was being done at IBM. Current SQL standards are ratified and published by the International Standards Organization (ISO). Although a new standard is published every few years, the last significant set of changes to the core language can be traced to the SQL:1999 standard (also known as "SQL3"). Subsequent standards have mainly been concerned with storing and processing XML-based data. Overall, the evolution of SQL is firmly rooted in the practical aspects of database development, and in many cases new standards only serve to ratify and standardize syntax or features that have been present in commercial database products for some time.

Declarative

The core of SQL is a *declarative language*. In a declarative language, you state what you want the results to be and allow the language processor to figure out how to deliver the desired results. Compare this to *imperative languages*, such as C, Java, Perl, or Python, where each step of a calculation or operation must be explicitly written out, leaving it up to the programmer to lead the program, step by step, to the correct conclusion.

The first SQL standards were specifically designed to make the language approachable and usable by "non-computer people"—at least by the 1980s definition of that term. This is one of the reasons why SQL commands tend to have a somewhat English-like

syntax. Most SQL commands take the form *verb-subject*. For example, `CREATE` (verb) `TABLE` (subject), `DROP INDEX`, `UPDATE` *`table_name`*.

The almost-English, declarative nature of SQL has both advantages and disadvantages. Declarative languages tend to be simpler (especially for nonprogrammers) to understand. Once you get used to the general structure of the commands, the use of English keywords can make the syntax easier to remember. The fixed command structure also makes it much easier for the database engine to optimize queries and return data more efficiently.

The predefined nature of declarative statements can sometimes feel a bit limited, however—especially in a command-based language like SQL, where individual, isolated commands are constructed and issued one at a time. If you require a query that doesn't fit into the processing order defined by the `SELECT` command, you may find yourself having to patch together nested queries or temporary tables. This is especially true when the problem you're trying to solve is inherently nonrelational, and you're forced to jump through extra hoops to account for that.

Despite its oddities and somewhat organic evolution, SQL is a powerful language with a surprisingly broad ability to express complex operations. Once you wrap your head around what makes SQL tick, you can often find moderately simple solutions to even the most off-the-beaten-path problems. It can take some adjustment, however, especially if your primary experience is with imperative languages.

Portability

SQL's biggest flaw is that formal standardization has almost always followed common implementations. Almost every database product (including SQLite) has custom extensions and enhancements to the core language that help differentiate it from other products, or expose features or control systems that are not covered by the core SQL standard. Often these enhancements are related to performance enhancements, and can be difficult to ignore.

While this less-strict approach to language purity has allowed SQL to grow and evolve in very practical ways, it means that "real world" SQL portability is not all that practical. If you strictly limit yourself to standardized SQL syntax, you can achieve a moderate degree of portability, but normally this comes at the cost of lower performance and less data integrity. Generally, applications will write to the specific SQL dialect they're using and not worry about cross-database compatibility. If cross-database compatibility is important to a specific application, the normal approach is to develop a core list of SQL commands required by the application, with minor tweaks for each specific database product.

SQLite makes an effort to follow the SQL standards as much as possible. SQLite will also recognize and correctly parse a number of nonstandard syntax conventions used by other popular databases. This can help with the portability issues.

SQL is not without other issues, but considering its lineage, it is surprisingly well suited for the task at hand. Love it or hate it, it is the relational database language of choice, and it is likely to be with us for a long time.

General Syntax

Before getting into specific commands in SQL, it is worth looking at the general language structure. Like most languages, SQL has a fairly complete expression syntax that can be used to define command parameters. A more detailed description of the expression support can be found in Appendix D.

Basic Syntax

SQL consists of a number of different commands, such as `CREATE TABLE` or `INSERT`. These commands are issued and processed one at a time. Each command implements a different action or feature of the database system.

Although it is customary to use all capital letters for SQL commands and keywords, SQL is a case-insensitive* language. All commands and keywords are case insensitive, as are identifiers (such as table names and column names).

Identifiers must be given as literals. If necessary, identifiers can be enclosed in the standards compliant double-quotes (`" "`) to allow the inclusion of spaces or other nonstandard characters in an identifier. SQLite also allows identifiers to be enclosed in square brackets (`[ ]`) or back ticks (`` ` ` ``) for compatibility with other popular database products. SQLite reserves the use of any identifier that uses `sqlite_` as a prefix.

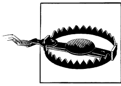
SQL is whitespace insensitive, including line breaks. Individual statements are separated by a semicolon. If you're using an interactive application, such as the `sqlite3` command-line tool, then you'll need to use a semicolon to indicate the end of a statement. The semicolon is not strictly required for single statements, however, as it is properly a statement separator and not a statement terminator. When passing SQL commands into the programming API, the semicolon is not required unless you are passing more than one command statement within a single string.

Single-line comments start with a double dash (`--`) and go to the end of the line. SQL also supports multi-line comments using the C comment syntax (`/* */`).

As with most languages, numeric literals are represented bare. Both integer (`453`) and real (rational) numbers (`43.23`) are recognized, as is exponent-style scientific notation (`9.745e-6`). In order to avoid ambiguities in the parser, SQLite requires that the decimal point is always represented as a period (`.`), regardless of the current internationalization setting.

* Unless otherwise specified, case insensitivity only applies to ASCII characters. That is, characters represented by values less than 128.

Text literals are enclosed in single quotes (`' '`). To represent a string literal that includes a single quote character, use two single quotes in a row (`publisher = 'O''Reilly'`). C-style backslash escapes (`\'`) are not part of the SQL standard and are not supported by SQLite. BLOB literals (binary data) can be represented as an `x` (or `X`) followed by a string literal of hexadecimal characters (`x'A554E59C'`).



Text literals use single quotes. Double quotes are reserved for identifiers (table names, columns, etc.). C-style backslash escapes are not part of the SQL standard.

SQL statements and expressions frequently contain lists. A comma is used as the list separator. SQL does not allow for a trailing comma following the last item of a list.

In general, expressions can be used any place a literal data value is allowed. Expressions can include both mathematical statements, as well as functions. Function-calling syntax is similar to most other computer languages, utilizing the name of the function, followed by a list of parameters enclosed in parentheses. Expressions can be grouped into subexpressions using parentheses.

If an expression is evaluated in the context of a row (such as a filtering expression), the value of a row element can be extracted by naming the column. You may have to qualify the column name with a table name or alias. If you're using cross-database queries, you may also have to specify which database you're referring to. The syntax is:

```
[[database_name.]table_name.]column_name
```

If no database name is given, it is assumed you're referring to the `main` database on the default connection. If the table name/alias is also omitted, the system will make a best-guess using just the column name, but will return an error if the name is ambiguous.

Three-Valued Logic

SQL allows any value to be assigned a *NULL*. NULL is not a value in itself (SQLite actually implements it as a unique valueless type), but is used as a marker or flag to represent unknown or missing data. The thought is that there are times when values for a specific row element may not be available or may not be applicable.

A NULL may not be a value, but it can be assigned to data elements that normally have values, and can therefore show up in expressions. The problem is that NULLs don't interact well with other values. If a NULL represents an unknown that might be any possible value, how can we know if the expression `NULL > 3` is true or false?

To deal with this problem, SQL must employ a concept called *three-valued logic*. Three-valued logic is often abbreviated *TVL* or *3VL*, and is more formally known as *ternary logic*. 3VL essentially adds an "unknown" state to the familiar true/false Boolean logic system.