Introduction

The Definitive Guide to SQLite covers SQLite in a comprehensive fashion, giving you the knowledge and experience to use it in a wide range of situations. Whether you are a hard-core C developer, are a mobile device aficionado, or are just seeking more know-how on the best embedded and small-footprint database engine ever invented, this book it for you.

Prerequisites

This book assumes no prior knowledge of SQLite, though naturally people of all experience levels will benefit from the material. SQLite is written in C with an extensive C API and also supports many other languages such as Python, Tcl, Ruby, and Java. As a database engine, it also makes extensive use of SQL. Although the examples in this book will benefit a reader of any skill level, we don't have space to also teach those languages.

How This Book Is Organized

The book contains 11 chapters, which cover the following broad areas:

- SQLite introduction, acquisition, and installation
- Using SQL with SQLite
- The C API for SQLite
- Using languages such as Python, Tcl, Ruby, and Java with SQLite
- Mobile device development with SQLite
- Internals and new features of SQLite

There's no real impediment to jumping in to whatever area takes your fancy, though you may find that Chapters 5, 6, and 7, which deal with the C API, are best approached in order.

Obtaining the Source Code of the Examples

The source code of all the examples in the book is available from the book's catalog page on the Apress web site, at `http://apress.com/book/view/1430232250`. Look for the "Source Code" link in the "Book Resources" sidebar.

CHAPTER 1

Introducing SQLite

SQLite is an open source, embedded relational database. Originally released in 2000, it is designed to provide a convenient way for applications to manage data without the overhead that often comes with dedicated relational database management systems. SQLite has a well-deserved reputation for being highly portable, easy to use, compact, efficient, and reliable.

An Embedded Database

SQLite is an *embedded* database. Rather than running independently as a stand-alone process, it symbiotically coexists inside the application it serves—within its process space. Its code is intertwined, or *embedded*, as part of the program that hosts it. To an outside observer, it would never be apparent that such a program had a relational database management system (RDBMS) on board. The program would just do its job and manage its data somehow, making no fanfare about how it went about doing so. But inside, there is a complete, self-contained database engine at work.

One advantage of having a database server inside your program is that no network configuration or administration is required. Take a moment to think about how liberating that is: no firewalls or address resolution to worry about, and no time wasted on managing intricate permissions and privileges. Both client and server run together in the same process. This reduces overhead related to network calls, simplifies database administration, and makes it easier to deploy your application. Everything you need is compiled right into your program.

Consider the processes found in Figure 1-1. One is a Perl script, another is a standard C/C++ program, and the last is an Apache-hosted PHP script, all using SQLite. The Perl script imports the DBI::SQLite module, which in turn is linked to the SQLite C API, pulling in the SQLite library. The PHP library works similarly, as does the C++ program. Ultimately, all three processes interface with the SQLite C API. All three therefore have SQLite embedded in their process spaces. By doing so, not only does each of those processes run their own respective code, but they've also become independent database servers in and of themselves. Furthermore, even though each process represents an independent server, they can still operate on the same database file(s) , benefitting from SQLite's use of the operating system to manage synchronization and locking.

Today there is a wide variety of relational database products on the market specifically designed for embedded use—products such as Sybase SQL Anywhere, Oracle TimesTen and BerkleyDB, Pervasive PSQL, and Microsoft's Jet Engine. Some of the dominant commercial vendors have pared down their large-scale databases to create embedded variants. Examples of these include IBM's DB2 Everyplace, Oracle's Database Lite, and Microsoft's SQL Server Express. The open source databases MySQL and Firebird both offer embedded versions as well. Of all these products, only one is open source, unencumbered by licensing fees, and designed exclusively for use as an embedded database: SQLite.

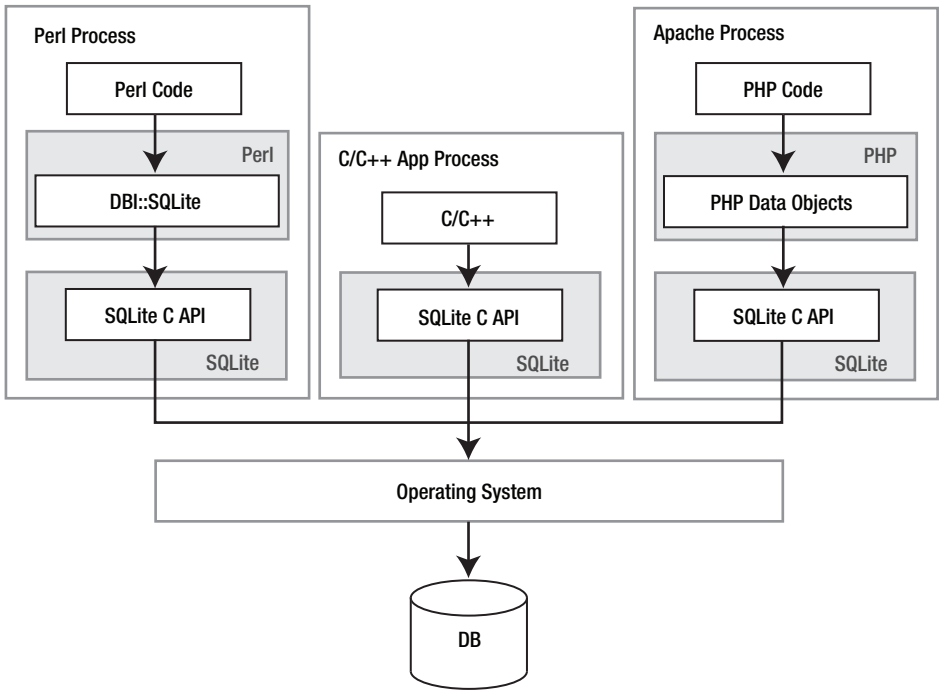

Figure 1-1. *SQLite embedded in host processes*

A Developer's Database

SQLite is quite versatile. It is a database, a programming library, and a command-line tool, as well an excellent learning tool that provides a good introduction to relational databases. There are many ways to use it—in embedded environments, websites, operating system services, scripts, and applications. For programmers, SQLite is like "data duct tape," providing an easy way to bind applications and their data. Like duct tape, there is no end to its potential uses. In a web environment, SQLite can help with managing complex session information. Rather than serializing session data into one big blob, individual pieces can be selectively written to and read from individual session databases. SQLite also serves as a good stand-in relational database for development and testing: there are no external RDBMSs or networking to configure or usernames and passwords to hinder the programmer's focus. SQLite can also serve as a cache, hold configuration data, or, by leveraging its binary compatibility across platforms, even work as an application file format.

Besides being just a storage receptacle, SQLite can serve as a purely functional tool for general data processing. Depending on size and complexity, it may be easier to represent some application data structures as a table or tables in an in-memory database. With so many developers, analysts, and others familiar with relational databases and SQL, you can benefit from "assumed knowledge"—operating on the data relationally by using SQLite to do the heavy lifting rather than having to write your own algorithms to manipulate and sort data structures. If you are a programmer, imagine how much code it would take to implement the following SQL statement in your program:

```
SELECT x, STDDEV(w)
FROM table
GROUP BY x
HAVING x > MIN(z) OR x < MAX(y)
ORDER BY y DESC
LIMIT 10 OFFSET 3;
```

If you are already familiar with SQL, imagine coding the equivalent of a subquery, compound query, `GROUP BY` clause, or multiway join in your favorite (or not so favorite) programming language. SQLite embeds all of this functionality into your application with minimal cost. With a database engine integrated directly into your code, you can begin to think of SQL as an offload engine in which to implement complex sorting algorithms in your program. This approach becomes more appealing as the size of your data set grows or as your algorithms become more complex. What's more, SQLite can be configured to use a fixed amount of RAM and then offload data to disk if it exceeds the specified limit. This is even harder to do if you write your own algorithms. With SQLite, this feature is available with a simple call to a single SQL command.

SQLite is also a great learning tool for programmers—a cornucopia for studying computer science topics. From parser generators to tokenizers, virtual machines, B-tree algorithms, caching, program architecture, and more, it is a fantastic vehicle for exploring many well-established computer science concepts. Its modularity, small size, and simplicity make it easy to present each topic as an isolated case study that any individual could easily follow.

An Administrator's Database

SQLite is not just a programmer's database. It is a useful tool for system administrators as well. It is small, compact, and elegant like finely honed versatile utilities such as `find`, `rsync`, and `grep`. SQLite has a command-line utility that can be used from the shell or command line and within shell scripts. However, it works even better with a large variety of scripting languages such as Perl, Python, TCL, and Ruby. Together the two can help with pretty much any task you can imagine, such as aggregating log file data, monitoring disk quotas, or performing bandwidth accounting in shared networks. Furthermore, since SQLite databases are ordinary disk files, they are easy to work with, transport, and back up.

SQLite is a convenient learning tool for administrators looking to learn more about relational databases. It is an ideal beginner's database with which to learn about relational concepts and practice their implementation. It can be installed quickly and easily on any platform you're likely to encounter, and its database files share freely between them without the need for conversion. It is full featured but not daunting. And SQLite—both the program and the database—can be carried around on a USB stick or memory chip.

SQLite History

SQLite was conceived on a battleship…well, sort of. SQLite's author, D. Richard Hipp, was working for General Dynamics on a program for the U.S. Navy developing software for use on board guided missile destroyers. The program originally ran on Hewlett-Packard Unix (HP-UX) and used an Informix database as the back end. For their particular application, Informix was somewhat overkill. For an experienced database administrator (DBA) at the time, it could take almost an entire day to install or upgrade. To the uninitiated application programmer, it might take forever. What was really needed was a self-contained database that was easy to use and that could travel with the program and run anywhere regardless of what other software was or wasn't installed on the system.

In January 2000, Hipp and a colleague discussed the idea of creating a simple embedded SQL database that would use the GNU DBM hash library (gdbm) as a back end, one that would require no installation or administrative support whatsoever. Later, when some free time opened up, Hipp started work on the project, and in August 2000, SQLite 1.0 was released.

As planned, SQLite 1.0 used `gdbm` as its storage manager. However, Hipp soon replaced it with his own B-tree implementation that supported transactions and stored records in key order. With the first major upgrade in hand, SQLite began a steady evolution, growing in both features and users. By mid-2001, many projects—both open source and commercial alike—started to use it. In the years that followed, other members of the open source community started to write SQLite extensions for their favorite scripting languages and libraries. One by one, new extensions for popular languages and APIs such as Open Database Connectivity (ODBC), Perl, Python, Ruby, Java, and others fell into place and testified to SQLite's wide application and utility.

SQLite began a major upgrade from version 2 to 3 in 2004. Its primary goal was enhanced internationalization supporting UTF-8 and UTF-16 text as well as user-defined text-collating sequences. Although version 3.0 was originally slated for release in summer 2005, America Online provided the necessary funding to see that it was completed by July 2004. Besides internationalization, version 3 brought many other new features such as a revamped C API, a more compact format for database files (a 25 percent size reduction), manifest typing, binary large object (BLOB) support, 64-bit ROWIDs, autovacuum, and improved concurrency. Even with many new features, the overall library footprint was still less than 240KB, at a time when most home PCs began measuring their memory in gigabytes! Another improvement in version 3 was a good code cleanup—revisiting, refactoring and rewriting, or otherwise throwing out the detritus accumulated in the 2.*x* series.

SQLite continues to grow feature-wise while still remaining true to its initial design goals: simplicity, flexibility, compactness, speed, and overall ease of use. At the time of this book's latest edition, SQLite has leapt ahead to include such advanced features as recursive triggers, distribution histograms to help the optimizer produce even faster queries, and asynchronous I/O on operating systems capable of supporting such workloads. What's next after that? Well, it all depends. Perhaps you or your company will sponsor the next big feature that makes this super-efficient database even better.

Who Uses SQLite

Today, SQLite is used in a wide variety of software and products. It is used in Apple's Mac OS X operating system, Safari web browser, Mail.app email program, and RSS manager, as well as Apple's Aperture photography software. Perhaps Apple's biggest use for SQLite has come in the iPhone age. You'll find many apps on the iPhone, such as the Contacts database, Notes, and more, all rely on SQLite. This reliance on SQLite also extends to the iPad. We'll return to this topic in Chapter 9, where we'll discuss working with SQLite on Apple's mobile platforms in detail.

SQLite can be found in Sun's Solaris operating environment, specifically the database backing the Service Management Facility that debuted with Solaris 10, a core component of its predictive self-healing technology. SQLite is in the Mozilla Project's mozStorage C++/JavaScript API layer, which will be the backbone of personal information storage for Firefox, Thunderbird, and Sunbird. SQLite has been added as part of the PHP 5 standard library. It also ships as part of Trolltech's cross-platform Qt C++ application framework, which is the foundation of the popular KDE window manager, and many other software applications. SQLite is especially popular in embedded platforms. Much of Richard Hipp's SQLite-related business has been porting SQLite to various proprietary embedded platforms. Symbian uses SQLite to provide SQL support in the native Symbian OS platform. Google has made extensive use of SQLite in the Android mobile phone operating system and user-space applications. SQLite is so pervasive within Android that Chapter 10 is dedicated to showing you all about its use on Android devices. SQLite is also included in commercial development products for cell phone applications.

Although it is rarely advertised, SQLite is also used in a variety of consumer products, as some tech-savvy consumers have discovered in the course of poking around under the hood. Examples include the D-Link Media Lounge, the Slim Devices Squeezebox music player, and the Philips GoGear personal music player. Some clever consumers have even found a SQLite database embedded in the *Complete New Yorker* DVD set—a digital library of every issue of the *New Yorker* magazine—apparently used by its accompanying search software.

You can find SQLite as an alternative back-end storage facility for a wide array of open source projects such as Yum (the package manager for Fedora Core), Movable Type, DSPAM, and Edgewall Software's excellent Trac SCM and project management system; possibly most famously, it's used as the built-in database for Firefox, the web browser from Mozilla. Even parts of SQLite's core utilities can be found in other open source projects. One such example is its Lemon parser generator, which the lighttpd web server project uses for generating the parser code for reading its configuration file. Indeed, there seems to be such a variety of uses for SQLite that Google took notice and awarded Richard Hipp with "Best Integrator" at O'Reilly's 2005 Open Source Convention—long before Google entered the mobile space with Android. See also `www.sqlite.org/famous.html` for other ideas of important SQLite users.

Architecture

SQLite has an elegant, modular architecture that takes some unique approaches to relational database management. It consists of eight separate modules grouped within three major subsystems (as shown in Figure 1-2). These modules divide query processing into discrete tasks that work like an assembly line. The top of the stack compiles the query, the middle executes it, and the bottom handles storage and interfacing with the operating system.

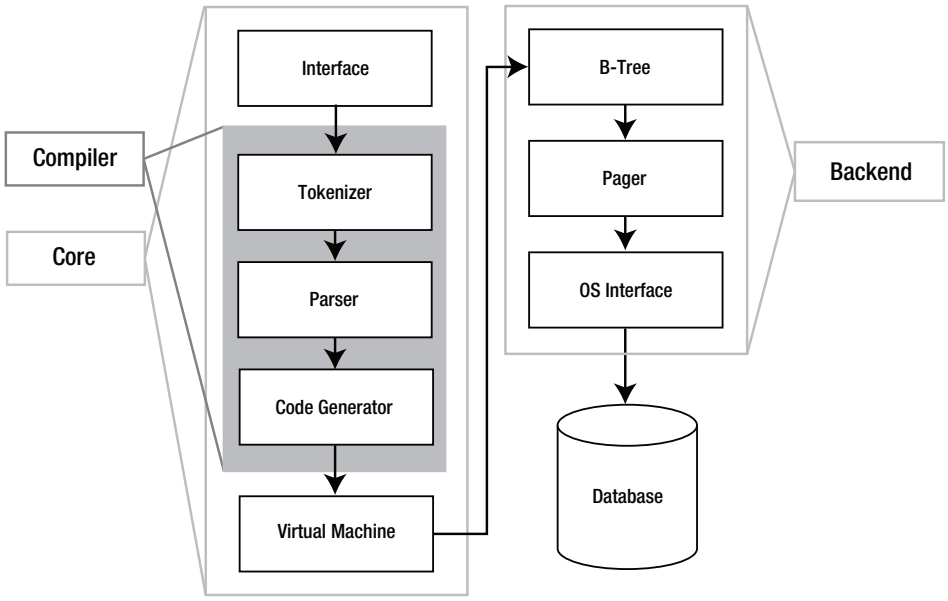

Figure 1-2. *SQLite's architecture*

The Interface

The interface is the top of the stack and consists of the SQLite C API. It is the means through which programs, scripting languages, and libraries alike interact with SQLite. Literally, this is where you as developer, administrator, student, or mad scientist talk to SQLite.

The Compiler

The compilation process starts with the tokenizer and parser. They work together to take a Structured Query Language (SQL) statement in text form, validate its syntax, and then convert it to a hierarchical data structure that the lower layers can more easily manipulate. SQLite's tokenizer is hand-coded. Its parser is generated by SQLite's custom parser generator, which is called Lemon. The Lemon parser generator is designed for high performance and takes special precautions to guard against memory leaks. Once the statement has been broken into tokens, evaluated, and recast in the form of a parse tree, the parser passes the tree down to the code generator.

The code generator translates the parse tree into a kind of assembly language specific to SQLite. This assembly language consists of instructions that are executable by its virtual machine. The code generator's sole job is to convert the parse tree into a complete mini-program written in this assembly language and to hand it off to the virtual machine for processing.

The Virtual Machine

At the center of the stack is the virtual machine, also called the *virtual database engine* (VDBE). The VDBE is a register-based virtual machine that works on byte code, making it independent of the underlying operating system, CPU, or system architecture. The VDBE's byte code (or virtual machine language) consists of more than 100 possible tasks known as *opcodes*, which are all centered on database operations. The VDBE is designed specifically for data processing. Every instruction in its instruction set either accomplishes a specific database operation (such as opening a cursor on a table, making a record, extracting a column, or beginning a transaction) or performs manipulations to prepare for such an operation. Altogether and in the right order, the VDBE's instruction set can satisfy any SQL command, however complex. Every SQL statement in SQLite—from selecting and updating rows to creating tables, views, and indexes—is first compiled into this virtual machine language, forming a stand-alone instruction set that defines how to perform the given command. For example, take the following statement:

```
SELECT name FROM episodes LIMIT 10;
```

This compiles into the VDBE program shown in Listing 1-1.

Listing 1-1. *VDBE Assembly*

```
addr        opcode      p1          p2          p3          p4          p5          comment
0           Trace       0           0           0                       00
1           Integer     10          1           0                       00
2           Goto        0           11          0                       00
3           OpenRead    0           2           0           3           00
4           Rewind      0           9           0                       00
5           Column      0           2           2                       00
6           ResultRow   2           1           0                       00
7           IfZero      1           9           -1                      00
8           Next        0           5           0                       01
9           Close       0           0           0                       00
10          Halt        0           0           0                       00
11          Transactio  0           0           0                       00
12          VerifyCook  0           4           0                       00
13          TableLock   0           2           0           episodes    00
14          Goto        0           3           0                       00
```

The program consists of 15 instructions. These instructions, performed in this particular order with the given operands, will return the `name` field of the first ten records in the `episodes` table (which is part of the example database included with this book).

In many ways, the VDBE is the heart of SQLite. All of the modules before it work to create a VDBE program, while all modules after it exist to execute that program, one instruction at a time.

The Back End

The back end consists of the B-tree, page cache, and OS interface. The B-tree and page cache (pager) work together as information brokers. Their currency is database pages, which are uniformly sized blocks of data that, like shipping containers, are made for transportation. Inside the pages are the goods: more interesting bits of information such as records and columns and index entries. Neither the B-tree nor the pager has any knowledge of the contents. They only move and order pages; they don't care what's inside.

The B-tree's job is order. It maintains many complex and intricate relationships between pages, which keeps everything connected and easy to locate. It organizes pages into tree-like structures (which is the reason for the name), which are highly optimized for searching. The pager serves the B-tree, feeding it pages. Its job is transportation and efficiency. The pager transfers pages to and from disk at the B-tree's behest. Disk operations are still some of the slowest things a computer has to do, even with today's solid-state disks. Therefore, the pager tries to speed this up by keeping frequently used pages cached in memory and thus minimizes the number of times it has to deal directly with the hard drive. It uses special techniques to predict which pages will be needed in the future and thus anticipate the needs of the B-tree, keeping pages flying as fast as possible. Also in the pager's job description are transaction management, database locking, and crash recovery. Many of these jobs are mediated by the OS interface.

Things such as file locking are often implemented differently in different operating systems. The OS interface provides an abstraction layer that hides these differences from the other SQLite modules. The end result is that the other modules see a single consistent interface with which to do things like file locking. So, the pager, for example, doesn't have to worry about doing file locking one way on Windows and doing it another way on different operating systems such as Unix. It lets the OS interface worry about this. It just says to the OS interface, "Lock this file," and the OS interface figures out how to do that

based on the operating system on which it happens to be running. Not only does the OS interface keep code simple and tidy in the other modules, but it also keeps the messy issues cleanly organized and at arm's length in one place. This makes it easier to port SQLite to different operating systems—all of the OS issues that must be addressed are clearly identified and documented in the OS interface's API.

Utilities and Test Code

Miscellaneous utilities and common services such as memory allocation, string comparison, and Unicode conversion routines are kept in the utilities module. This is basically a catchall module for services that multiple modules need to use or share. The testing module contains a myriad of regression tests designed to examine every little corner of the database code. This module is one of the reasons SQLite is so reliable: it performs a lot of regression testing and makes those tests available for anyone to run and improve.

SQLite's Features and Philosophy

SQLite offers a surprisingly comprehensive range of features and capabilities despite its small size. It supports a large subset of the ANSI SQL92 standard for SQL features (transactions, views, check constraints, foreign keys, correlated subqueries, compound queries, and more) along with many other features found in relational databases, such as triggers, indexes, autoincrement columns, and `LIMIT/OFFSET` features. It also has many rare or unique features, such as in-memory databases, dynamic typing, and conflict resolution—otherwise referred to as `merge` or `upsert` in other RDBMSs—which will be explained in a moment.

As mentioned earlier in this chapter, SQLite has a number of governing principles or characteristics that serve to more or less define its philosophy and implementation. I'll expand on these issues next.

Zero Configuration

From its initial conception, SQLite has been designed so that it can be incorporated and used without the need of a DBA. Configuring and administering SQLite is as simple as it gets. SQLite contains just enough features to fit in a single programmer's brain, and like its library, it requires as small a footprint in the gray matter as it does in RAM.

Portability

SQLite was designed specifically with portability in mind. It compiles and runs on Windows, Linux, BSD, Mac OS X, and commercial Unix systems such as Solaris, HP-UX, and AIX, as well as many embedded platforms such as QNX, VxWorks, Symbian, Palm OS, and Windows CE. It works seamlessly on 32- and 64-bit architectures with both big- and little-endian byte orders. Portability doesn't stop with the software either: SQLite's database files are as portable as its code. The database file format is binary compatible across all supported operating systems, hardware architectures, and byte orders. You can create a SQLite database on a Linux workstation and use it on a Mac or Windows machine, an iPhone, or other device, without any conversion or modification. Furthermore, SQLite databases can hold up to 2 terabytes of data (limited only by the operating system's maximum file size) and natively support both UTF-8 and UTF-16 encoding.

Compactness

SQLite was designed to be lightweight and self-contained; one header file, one library, and you're relational—no external database server required. Everything packs into less than half a megabyte, which is smaller than many of the web pages you'll visit on any given day.

SQLite databases are ordinary operating system files. Regardless of your system, all objects in your SQLite database—tables, triggers, schema, indexes, and views—are contained in a single operating system file. SQLite uses variable-length records wherever possible, allocating only the minimum amount of data needed to hold each field. A 2-byte field sitting in a `varchar(100)` column takes up only 3 bytes of space, not 100 (the extra byte is used to record its type information).

Simplicity

As a programming library, SQLite's API is one of the simplest and easiest to use. The API is both well documented and intuitive. It is designed to help you customize SQLite in many ways, such as implementing your own custom SQL functions in C. The open source community also has a created a vast number of language and library interfaces with which to use SQLite. There are extensions for Perl, Python, Ruby, Tcl/Tk, Java, PHP, Visual Basic, ODBC, Delphi, C#, VB .NET, Smalltalk, Ada, Objective C, Eiffel, Rexx, Lisp, Scheme, Lua, Pike, Objective Camel, Qt, WxWindows, REALBASIC, and others. You can find an exhaustive list on the SQLite wiki: `www.sqlite.org/cvstrac/wiki?p=SqliteWrappers`.

SQLite's modular design includes many innovative ideas that enable it to be full featured and extensible while at the same time retaining a great degree of simplicity throughout its code base. Each module is a specialized, independent system that performs a specific task. This modularity makes it much easier to develop each system independently and to debug queries as they pass from one module to the next—from compilation and planning to execution and materialization. The end result is that there is a crisp, well-defined separation between the front end (SQL compiler) and back end (storage system), allowing the two to be coded independently of each other. This design makes it easier to add new features to the database engine, is faster to debug, and results in better overall reliability.

Flexibility

Several factors work together to make SQLite a very flexible database. As an embedded database, it offers the best of both worlds: the power and flexibility of a relational database front end, with the simplicity and compactness of a B-tree back end. With it, there are no large database servers to configure, no networking or connectivity problems to worry about, no platform limitations, and no license fees or royalties to pay. Rather, you get simple SQL support dropped right into your application.

Liberal Licensing

All of SQLite's code is in the public domain. There is no license. No claim of copyright is made on any part of the core source code. All contributors to this code are required to sign affidavits specifically disavowing any copyright interest in contributed code. Thus, there are no legal restrictions on how you may use the source code in any form. You can modify, incorporate, distribute, sell, and use the code for any purpose—commercial or otherwise—without any royalty fees or restrictions.

Reliability

The SQLite source code is more than just free; it also happens to be well written. SQLite's code base consists of about 70,000 lines of standard ANSI C that are clean, modular, and well commented. The code base is designed to be approachable, easy to understand, easy to customize, and generally very accessible. It is easily within the ability of a competent C programmer to follow any part of SQLite or the whole of it with sufficient time and study.

Additionally, SQLite's code offers a full-featured API specifically for customizing and extending SQLite through the addition of user-defined functions, aggregates, and collating sequences along with support for operational security.

While SQLite's modular design significantly contributes to its overall reliability, its source code is also well tested. Whereas the core software (library and utilities) consists of about 70,000 lines of code, the distribution includes an extensive test suite consisting of more than 45 *million* lines of test code, which as of July 2009 covers 100 percent of the core code. Ask any developer how hard it is to be that comprehensive when testing a nontrivial amount of code, and you can see why people have rock-solid confidence in the reliability of SQLite.

Convenience

SQLite also has a number of unique features that provide a great degree of convenience. These include dynamic typing, conflict resolution, and the ability to "attach" multiple databases to a single session.

SQLite's dynamic typing is somewhat akin to that found in scripting languages (e.g., *duck typing* in Ruby). Specifically, the type of a variable is determined by its value, not by a declaration as employed in statically typed languages. Most database systems restrict a field's value to the type declared in its respective column. For example, each field in an integer column can hold only integers or possibly NULL. In SQLite, while a column can have a declared type, fields are free to deviate from them, just as a variable in a scripting language can be reassigned a value with a different type. This can be especially helpful for prototyping, since SQLite does not force you to explicitly change a column's type. You need only change how your program stores information in that column rather than continually having to update the schema and reload your data.

Conflict resolution is another great feature. It can make writing SQL, as easy as it is, even easier. This feature is built into many SQL operations and can be made to perform what can be called *lazy updates*. Say you have a record you need to insert, but you are not sure whether one just like it already exists in the database. Rather than write a `SELECT` statement to look for a match and then recast your `INSERT` to an `UPDATE` if it does, conflict resolution lets you say to SQLite, "Here, try to insert this record, and if you find one with the same key, just update it with these values instead." Now you've gone from having to code three different SQL statements to cover all the bases (i.e., `SELECT`, `INSERT`, and possibly `UPDATE`) to just one: `INSERT OR REPLACE` (...). Other relational database systems mimic this aspect of conflict resolution with `UPSERT` or `MERGE` statements, but SQLite goes one step further. You can build conflict resolution into the table definition itself and dispense with the need to ever specify it again on future `INSERT` statements. In fact, you can dispense with ever having to write `UPDATE` statements to this table again—just write `INSERT` statements and let SQLite do the dirty work of figuring out what to do using the conflict resolution rules defined in the schema.

Finally, SQLite lets you "attach" external databases to your current session. Say you are connected to one database (`foo.db`) and need to work on another (`bar.db`). Rather than opening a separate connection and juggling multiple connection handles in your code, you can simply attach the database of interest to your current connection with a single SQL command:

```
ATTACH database bar.db as bar;
```

All of the tables in `bar.db` are now accessible as if they existed in `foo.db`. You can detach it just as easily when you're done. This makes all sorts of things like copying tables between databases very easy.

Performance and Limitations

SQLite is a speedy database. But the words *speedy*, *fast*, *peppy*, or *quick* are rather subjective terms. To be perfectly honest, there are things SQLite can do faster than other databases, and there are things that it cannot. Suffice it to say, within the parameters for which it has been designed, SQLite can be said to be consistently fast and efficient across the board. SQLite uses B-trees for indexes and B+-trees for tables, the same as most other database systems. For searching a single table, it is as fast if not faster than any other database on average. Simple `SELECT`, `INSERT`, and `UPDATE` statements are extremely quick—virtually at the speed of RAM (for in-memory databases) or disk. Here SQLite is often faster than other databases, because it has less overhead to deal with in starting a transaction or generating a query plan and because it doesn't incur the overhead of making a network calls to the server or negotiating authentication and privileges. Its simplicity here makes it fast.

As queries become larger and more complex, however, query time overshadows the network call or transaction overhead, and the game goes to the database with the best optimizer. This is where larger, more sophisticated databases begin to shine. While SQLite can certainly do complex queries, it does not have a sophisticated optimizer or query planner. You can always trust SQLite to give you the result, but what it won't do is try to determine optimal paths by computing millions of alternative query plans and selecting the fastest candidate, as you might expect from Oracle or PostgreSQL. Thus, if you are running complex queries on large data sets, odds are that SQLite is not going to be as fast as databases with sophisticated optimizers.

So, there are situations where SQLite is not as fast as larger databases. But many if not all of these conditions are to be expected. SQLite is an embedded database designed for small to medium-sized applications. These limitations are in line with its intended purpose. Many new users make the mistake of assuming that they can use SQLite as a drop-in replacement for larger relational databases. Sometimes you can; sometimes you can't. It all depends on what you are trying to do.

In general, there are two major variables that define SQLite's main limitations:

- **Concurrency.** SQLite has coarse-grained locking, which allows multiple readers but only one writer at a time. Writers exclusively lock the database during writes, and no one else has access during that time. SQLite does take steps to minimize the amount of time in which exclusive locks are held. Generally, locks in SQLite are kept for only a few milliseconds. But as a general rule of thumb, if your application has high write concurrency (many connections competing to write to the same database) and it is time critical, you probably need another database. It is really a matter of testing your application to know what kind of performance you can get. We have seen SQLite handle more than 500 transactions per second for 100 concurrent connections in simple web applications. But your transactions may differ in the number of records being modified or the number and complexity of the queries involved. Acceptable concurrency all depends on your particular application and can be determined empirically only by direct testing. In general, this is true with any database: you don't know what kind of performance your application will get until you do real-world tests.

- **Networking.** Although SQLite databases can be shared over network file systems, the latency associated with such file systems can cause performance to suffer. Worse, bugs in network file system implementations can also make opening and modifying remote files—SQLite or otherwise—error prone. If the file system's locking does not work properly, two clients may be allowed to simultaneously modify the same database file, which will almost certainly result in database corruption. It is not that SQLite is incapable of working over a network file system because of anything in its implementation. Rather, SQLite is at the mercy of the underlying file system and wire protocol, and those technologies are not always perfect. For instance, many versions of NFS have a flawed `fcntl()` implementation, meaning that locking does not behave as intended. Newer NFS versions, such are Solaris NFS v4, work just fine and reliably implement the requisite locking mechanisms needed by SQLite. However, the SQLite developers have neither the time nor the resources to certify that any given network file system works flawlessly in all cases.

Again, most of these limitations are intentional, resulting from SQLite's design. Supporting high write concurrency, for example, brings with it great deal of complexity, and this runs counter to SQLite's simplicity in design. Similarly, being an embedded database, SQLite intentionally does not support networking. This should come as no surprise. In short, what SQLite can't do is a direct result of what it can. It was designed to operate as a modular, simple, compact, and easy-to-use embedded relational database whose code base is within the reach of the programmers using it. And in many respects, it can do what many other databases cannot, such as run in embedded environments where actual *power consumption* is a limiting factor.

While SQLite's SQL implementation is quite good, there are some things it currently does not implement. These are as follows:

- **Complete trigger support.** SQLite supports almost all the standard features for triggers, including recursive triggers and `INSTEAD OF` triggers. However, for all trigger types, SQLite currently requires `FOR EACH ROW` behavior, where the triggered action is evaluated for every row affected by the triggering query. ANSI SQL 92 also describes a `FOR EACH STATEMENT` trigger style, which is not currently supported.

- **Complete ALTER TABLE support.** Only the `RENAME TABLE` and `ADD COLUMN` variants of the `ALTER TABLE` command are supported. Other kinds of `ALTER TABLE` operations such as `DROP COLUMN`, `ALTER COLUMN`, and `ADD CONSTRAINT` are not implemented.

- **RIGHT and FULL OUTER JOIN.** `LEFT OUTER JOIN` is implemented, but `RIGHT OUTER JOIN` and `FULL OUTER JOIN` are not. Every `RIGHT OUTER JOIN` has a provably semantically identical `LEFT OUTER JOIN`, and vice versa. Any `RIGHT OUT JOIN` can be implemented as a `LEFT OUTER JOIN` by simply reversing the order of the tables and modifying the join constraint. A `FULL OUTER JOIN` can be implemented as a combination of two `LEFT OUTER JOIN` statements, a `UNION`, and appropriate `NULL` filtering in the `WHERE` clause.

- **Updatable views.** Views in SQLite are read-only. You may not execute a `DELETE`, `INSERT`, or `UPDATE` statement on a view. But you can create a trigger that fires on an attempt to `DELETE`, `INSERT`, or `UPDATE` a view and do what you need in the body of the trigger.

- **Windowing functions.** One of the new feature sets specified in ANSI SQL 99 are the windowing functions. These provide post-processing analytic functions for results, such a ranking, rolling averages, lag and lead calculation, and so on. SQLite currently only targets ANSI SQL 92 compliance, so it doesn't support windowing functions like `RANK()`, `ROW_NUMBER()`, and so on.
- **GRANT and REVOKE.** Since SQLite reads and writes an ordinary disk file, the only access permissions that can be applied are the normal file access permissions of the underlying operating system. `GRANT` and `REVOKE` commands in general are aimed at much higher-end systems where there are multiple users who have varying access levels to data in the database. In the SQLite model, the application is the main user and has access to the entire database. Access in this model is defined at the application level—specifically, what applications have access to the database file.

In addition to what is listed here, there is a page on the SQLite wiki devoted to reporting unsupported SQL. It is located at `www.sqlite.org/cvstrac/wiki?p=UnsupportedSql`.

Who Should Read This Book

SQLite has many uses and therefore draws a wide and diverse audience. Whether you are a programmer, web developer, systems administrator, or just casual user looking to learn about relational databases, this book aims to help you understand and get the most out of your particular use for SQLite.

SQLite is a terrific database to start on if you are new to relational databases. This book will help you not only get started with SQLite but also become a competent user of SQL. It will also provide you with a good foundation with which to move on to larger relational systems and explore more advanced features and topics.

For programmers, this book assumes only that you know the programming language in which you intend to use SQLite and the basics of relational theory. Furthermore, it does more than document APIs. If anything, that is the least of what it does, because API documentation only illustrates how an interface works. As with any database, you have to have some idea of how that database works internally to get the most out of it. Every database has unique architectural aspects, specific relational features, and important limitations, all of which good programmers learn about and take into consideration when writing their code. SQLite, though simple and straightforward, is no exception. As a programmer, you need to know something about how it processes data internally to get it to work well with your application. This book shows you how. It covers the API and explores how it works in relation to SQLite's architecture, allowing C programmers, web developers, and scriptwriters alike to write more informed code. This helps you better understand not only what SQLite can do but also what it can't. Your knowledge of the architecture will tell you better than any list of rules when SQLite is or isn't a good fit for what you are trying to accomplish. You'll know if, when, and where you need to consider another approach.

One of the most important aims in this book is to teach concepts over recipes—to adequately address both how and why. By the time you're done, you will have both a conceptual and practical understanding of how something works. To that end, this book includes both theoretical material intermixed with many figures and examples designed specifically to illustrate the topics at hand and provide real-world value.

How This Book Is Organized

This book is divided into three parts: SQLite the database, SQLite the programming library, and reference material. The database aspects of SQLite are covered in Chapters 2, 3, and 4. The programming aspects of SQLite are covered in Chapters 5–8. A brief chapter outline is as follows:

Chapter 1, "Introducing SQLite," introduces the main features of SQLite, its origin and history, and the scope and objectives of this book (you are reading this now!)

Chapter 2, "Getting Started," covers how to obtain and use SQLite. It illustrates how to get SQLite in binary and source form, as well as how to compile and build it on a variety of platforms. It explains how to use the SQLite command-line utility to create and work with databases.

Chapter 3, "SQL for SQLite," provides some background behind SQL. It introduces many of the basic SQL commands, such as those for creating tables and selecting and working with data. This chapter acts as both an overview for SQLite's own approach to SQL and a quick introduction for those needing a refresher on SQL itself.

Chapter 4, "Advanced SQL in SQLite," continues from Chapter 3 with coverage of more complex queries and explores every aspect of the remaining commands in SQLite's SQL implementation.

Chapter 5, "SQLite Design and Concepts," lays the groundwork for programming with SQLite. It illustrates the SQLite API, its architecture, and how the two work in relation to one another. It addresses topics such as transactions and locking. It provides programmers of all languages with a clear understanding of how SQLite works internally and what to keep in mind when writing programs that use it.

Chapter 6, "The Core C API," covers the part of the SQLite C API related to executing queries. With sections on connecting to databases, executing queries, obtaining data, managing transactions, and tracing your code, this chapter covers all parts of the API related to query and data processing.

Chapter 7, "The Extension C API," covers the remaining part of the C API devoted to customizing and extending SQLite. SQLite provides facilities for implementing user-defined SQL functions, aggregates, and collations. This chapter illustrates example implementations of these features and provides practical examples of their use.

Chapter 8, "Language Bindings for SQLite," provides a concise introduction to SQLite programming in a range of popular languages such as Perl, Python, Ruby, Java, and PHP.

Chapter 9, "SQLite for Apple iPhone and iPad," focuses on how to use SQLite when developing for Apple's mobile platforms.

Chapter 10, "SQLite for Google Android devices," covers how SQLite is baked in to the Android platform and how to develop with SQLite for the explosion of devices that now run Android.

Chapter 11, "SQLite Internals," explores the inner workings of SQLite. It is a high-level overview of the source code and provides a glimpse into how the major subsystems are implemented. This provides programmers with a deeper understanding of SQLite's design decisions, assumptions, and trade-offs, as well as a point of departure for developers who want to work on SQLite.

DATABASE EXAMPLES

The example databases accompanying this book are available online and can be downloaded from the Apress website (`www.apress.com`). Each database is in SQL format, and you can simply follow the procedures covered in Chapter 2 to create them using the SQLite command-line program. The example databases are further explained and illustrated as they are introduced in this book.

The source code for all examples is also available online. All examples compile and run on Windows, Linux, and Unix. For each example, makefiles are included for Linux and Unix environments, and Visual C++ projects have been created for Windows users.

Additional Information

The SQLite website has a wealth of information, including the official documentation, mailing lists, wiki, and other general information. It is located at `www.sqlite.org`. The SQLite community is very helpful, and you may find everything you need on SQLite's mailing list. Additionally, SQLite's author offers professional training and support for SQLite, which includes custom programming (porting to embedded platforms, etc.), and enhanced versions of SQLite, which include native encryption and extremely small versions optimized for embedded applications. You can find more information at `www.hwaci.com/sw/sqlite/prosupport.html`.

Summary

SQLite is not to be confused with other larger databases like Oracle or PostgreSQL. Whereas dedicated relational databases such as these are electronic juggernauts, SQLite is a digital Swiss Army knife. Whereas large-scale dedicated relational databases are designed for thousands of users, SQLite is designed for thousands of uses. It is more than a database. Although a tool in its own right, it is a tool for making tools as well. It is a true utility, engineered to enable you—the developer, user, or administrator—to quickly and easily shape those disparate piles of data into order, and manipulate them to your liking with minimal effort.

SQLite is public domain software. Free. You can do anything with it or its source code you like. No licenses, no install programs, no restrictions. Just copy and run. It is also portable, well tested, and reliable. It has a clean, modular design that helps keep the system simple, easy to develop, and easy to debug. In addition to good design, it has good testing. There is more code written to test SQLite than there is SQLite code to test. It should not be too surprising, then, that SQLite has proven itself to be a solid, reliable database over its five-year history.

Finally, SQLite is fun. At least we think so, and we hope that you will find it equally useful and enjoyable.

If you have any comments or suggestions on this book or its examples, please feel free to contact the authors at `sqlitebook@gmail.com` (Michael) or `grantondata@gmail.com` (Grant).