SQLite
By Chris Newman

Publisher: Sams Publishing
Pub Date: November 09, 2004
ISBN: 0-672-32685-X
Pages: 336

- Table of Contents
- Index

SQLite is a small, fast, embeddable database. What makes it popular is the combination of the database engine and interface into a single library as well as the ability to store all the data in a single file. Its functionality lies between MySQL and PostgreSQL, however it is faster than both databases.

In SQLite, author Chris Newman provides a thorough, practical guide to using, administering and programming this up-and-coming database. If you want to learn about SQLite or about its use in conjunction with PHP this is the book for you.

SQLite
By Chris Newman

Publisher: Sams Publishing
Pub Date: November 09, 2004
ISBN: 0-672-32685-X
Pages: 336

- Table of Contents
- Index

Copyright

Library of Congress Catalog Card Number: 2004901350

Printed in the United States of America

First Printing: November 2004

07 06 05 04 4 3 2 1

Trademarks

Warning and Disclaimer

Bulk Sales

Credits

Acquisitions Editor

Shelley Johnston

Development Editor

Foreword

When I first began composing SQLite in late May of 2000, I never imagined that a few years later it would be widely used in consumer electronics devices and in countless programs and that people I barely know would be writing books about it. Like most other open-source software, SQLite grew out of a personal need and was intended primarily for my own use. In January of 2000, I was working with a team from General Dynamics on a software project that was connected to an Informix database. When Informix works, it works well, but we were having problems getting it to start reliably when the host computer was rebooted. We substituted PostgreSQL for Informix on our development systems, but even that database was a hassle to administer. While I was struggling to deal with these issues, the idea arose to write a simple SQL database engine that was serverless, read ordinary disk files, and could be statically linked into the application. I took no action on the idea then. But five months later, I was without a contract for a few months and so I started writing SQLite with the thought that it would be handy the next time a similar problem appeared. The basic database engine was running within a day and was stable within a couple of weeks. Development was interrupted for a month when my wife, Ginger, and I traveled to central Africa to visit friends. After our return and exactly four years ago from the day that I write this, I posted SQLite version 1.0 on the web.

Version 1 of SQLite attracted a small number of users. But the library was of limited utility because it employed GDBM as a storage back end. GDBM has no transaction support and it is based on hashing, so indices could not be used to optimize queries that are constrained by inequalities. To address these limitations, I began working in my spare time on a replacement B-Tree back end in the spring of 2001. Several months passed. I remember that I was fixing some last-minute bugs in the new system one morning when Ginger called from work to tell me that an airplane had just crashed into the World Trade Center in New York. Version 2.0 was released a couple of weeks later, amid a zeitgeist of sadness.

The release of SQLite version 2.0 triggered a surge of interest. Within weeks, Christian Werner published the first SQLite add-on, an ODBC driver. Dozens of other wrappers would soon follow. Over the next 12 months, support was added for INTEGER PRIMARY KEY, for VIEWs, and LEFT OUTER JOINs. Dan Kennedy contributed code to implement trIGGERs. By the fall of 2002, SQLite was in essentially the same form as you find it today. All through that year and since, more and more people began using SQLite in their programs and products. In recent months, the SQLite website has received visitors from around three thousand distinct IP addresses per day. Source code downloads average more than 400 per day with almost twice that many binary downloads. Total website traffic is approaching one gigabyte per day. SQLite bindings now exist for over two dozen languages including Perl, Python, PHP, Tcl/Tk, Ruby, Lisp, and Java. SQLite has been incorporated into many popular open-source projects, such as Kexi, monotone, Mozilla, and Popfile to name a few. I have been told, in confidence, of many commercial software projects built around the library. SQLite has also been spotted in consumer electronics devices such the Philips HDD060 MP3 player and in the D-Link DSM-320 Wi-Fi Media Player. No doubt countless other uses of SQLite have escaped my notice.

From its inception, the primary design goal of SQLite has been simplicity. I have tried to keep SQLite simple to administer, simple to operate, simple to program, and simple to maintain and customize. Many people tell me that they like SQLite because it is small, fast, and reliable. Reliability is a consequence of simplicitywith less complication, there is less to go wrong. Small size and fast performance are just happy accidents. Simplicity is the ultimate goal. In this way, SQLite is different from enterprise-class database engines that give you most everything you could ever want, except for simplicity. SQLite may have fewer features, but it is much simpler to use and operate, which is more important than a rich feature set in many situations. This is not to say that SQLite will not add new features over time. SQLite will continue to advance and growwe are currently looking at adding support for ALTER TABLE and for foreign keys, for examplebut as long as I control its development, SQLite will continue to be as simple as I can make it.

Except for a few lines here and there, most of the code in SQLite was written by me and Dan Kennedy. But we have not been working in isolation. Suggestions and criticisms from the SQLite user community have been invaluable in helping to direct the library's progress. And though no outside code has been copied into SQLite, other software has been essential in the process of building and testing SQLite. Special thanks go to John Ousterhout and his Tcl scripting language. We would have never been able to get SQLite working had it not been for the Tcl language, which

About the Author

Chris Newman is a consultant programmer specializing in the development of custom web-based database applications to a loyal international client base.

A graduate of Keele University, Chris lives in Stoke-on-Trent, England, where he runs Lightwood Consultancy Ltd, the company he founded in 1999 to further his interest in Internet technology. Lightwood operates web hosting services under the DataSnake brand and is proud to be one of the first hosting companies to offer and support SQLite as a standard feature on all accounts.

More information on Lightwood Consultancy Ltd can be found at http://www.lightwood.net, and Chris can be contacted at chris@lightwood.net.

We Want to Hear from You!

As the reader of this book, you are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write me directly to let me know what you did or didn't like about this bookas well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or email address. I will carefully review your comments and share them with the author and editors who worked on the book.

Email:	opensource@samspublishing.com
Mail:	Mark Taber Associate Publisher Sams Publishing 800 East 96th Street Indianapolis, IN 46240 USA

Reader Services

For more information about this book or another Sams title, visit our Web site at www.samspublishing.com. Type the ISBN (excluding hyphens) or the title of a book in the Search field to find the page you're looking for.

Introduction
Welcome to SQLite

SQLite is one of the fastest-growing database engines around, but that's growth in terms of popularity, not anything to do with its size. In fact one of SQLite's greatest strengths is that it is extremely lightweight indeed yet still manages to retain a large number of features.

Why Use SQLite?

There are many reasons for choosing SQLite, including

- Performance SQLite performs database operations efficiently and is faster than other free databases such as MySQL and PostgreSQL.
- Size SQLite has a small memory footprint and only a single library is required to access databases, making it ideal for embedded database applications.
- Portability SQLite runs on many platforms and its databases can be ported easily with no client/server setup or ongoing administration required.
- Stability SQLite is ACID-compliant, meeting all four criteriaAtomicity, Consistency, Isolation, and Durability.
- SQL support SQLite implements a large subset of the ANSI-92 SQL standard, including views, subqueries, and triggers.
- Interfaces SQLite has language APIs for C/C++, PHP, Perl, Python, Tcl, and many more beyond those covered in this book.
- Cost SQLite is in the public domain and therefore is free to use for any purpose without cost and can be freely redistributed.

Who This Book Is For

This book is aimed at intermediate- to advanced-level programmers looking to include database functionality within their applications. You should have at least a basic working knowledge of one of the languages covered by this bookC/C++, PHP, Perl, Python, or Tcl. The underlying library is written in C/C++; however, it is not necessary to understand that language in order to use the full capabilities of SQLite in your applications.

If you are a new programmer you should still be able to pick up SQLite fairly quickly, but this book does not cover programming basics in any of the languages for which there is a SQLite interface. The benefits of SQLite are only realized through using a programming API as it does not include the tools required to operate as a standalone database system.

It is not a prerequisite to have used a relational database in the past. If SQLite will be the first SQL-based database you have encountered, the book gives a thorough SQL tutorial covering the syntax as understood by SQLite. SQL veterans will still benefit from reviewing the sections that cover features included and omitted by SQLite.

How the Book Is Organized

This book is organized into three parts.

Part I General SQLite Use

- Chapter 1: Getting Started Gives some background on SQLite and discusses its strengths and weaknesses. Covers a few basic SQL commands to get things going and looks at the interactive interfaces to SQLite.
- Chapter 2: Working with Data A concise SQLite tutorial introduces working with the SQLite back end. Discusses programming and database design issues related to SQLite.
- Chapter 3: SQLite Syntax and Use Examines SQL syntax as supported by SQLite and suggests workarounds for the unsupported features.
- Chapter 4: Query Optimization Discusses performance considerations related to SQL queries and gives some techniques that can be used to speed up your database application.

Part II Using SQLite Programming Interfaces

- Chapter 5: The PHP Interface How to use the SQLite library within PHP scripts to create dynamic, database-driven web pages.
- Chapter 6: The C/C++ Interface How to write C/C++ programs using the SQLite library.
- Chapter 7: The Perl Interface How to write Perl scripts using the Database Interface module and SQLite Database Driver.
- Chapter 8: The Tcl Interface How to write Tcl scripts using the supplied SQLite extension.
- Chapter 9: The Python Interface How to write Python programs using the PySQLite extension.

Part III SQLite Administration

- Chapter 10: General Database Administration Discusses basic administration of SQLite and examines SQLite's internal architecture and the Virtual Database Engine (VDBE).

Versions of Software Covered

At the time of writing, the most recent stable version of SQLite is 2.8.15; however, SQLite 3 is already available as a beta and includes some changes and enhancements over the version 2 series. This book was written with the established, stable, and well-supported version 2 series in mind.

For the other software discussed in this book, the current versions are as follows:

- PHP 5.0.1
- Perl 5.8.5
- Perl::DBI 1.4.3
- DBD::SQLite2 0.32
- Tcl 8.4.7
- Python 2.3.4

Additional Resources

The following are the primary web pages for each of the packages used in this book:

- SQLite http://www.sqlite.org/
- PHP http://www.php.net/
- Perl DBI http://dbi.perl.org/
- Tcl http://www.tcl.tk/
- Python http://www.python.org/

For support with any of these technologies the relevant mailing list is a good place to start. Instructions on joining them can be found in the following locations:

- SQLite http://www.sqlite.org/support.html
- PHP http://www.php.net/mailing-lists.php
- Perl DBI http://dbi.perl.org/support/
- Tcl http://wiki.tcl.tk/1301
- Python http://www.python.org/community/lists.html

Part I: General SQLite Use

Chapter 1. Getting Started

Welcome to SQLite. This chapter will give you an overview of what SQLite is and isn't, what it can and can't do, and how it compares to other databases that you might consider using or might already be familiar with.

We will look at a few basic SQL commands and see how to use the sqlite command-line tool and the SQLite Database Browser GUI to create, examine, and modify databases.

Introduction

SQLite is an embeddable SQL-driven database engine that implements both the database engine and its interface as a C/C++ library. Started in 2000 by D. Richard Hipp, it was written from the ground up and contains absolutely no legacy code, and the SQLite source code has been in the public domain since the first prerelease of version 2.0 in 2001.

The primary design goals when SQLite was conceived were that it should be

- Simple to administer
- Simple to operate
- Simple to use in a program
- Simple to maintain and customize

The fact that SQLite is small, fast, and reliablearguably its greatest strengthsis, according to Hipp, a happy coincidence. He concentrated on making SQLite simple, and reliability is a byproduct of having fewer things to go wrong. Having simpler code in the database engine makes it much easier to optimize.

Note

The acronym SQL is sometimes pronounced sequel, although in common usage it is most often said as three letters. SQLite, however, is pronounced sequel-lite by its creatorin the same way that Microsoft SQL Server is usually pronounced sequel-serverand therefore that is how we have assumed it is said in this book. As we will refer to a SQLite database and an SQL statement, it will help if you are used to hearing them this way as you read on.

SQLite is already widely used in many projects, and its popularity looks set to continue growing. At the time of writing, version 3 of SQLite is close to completion and a stable release is expected to be available by the end of 2004.

Note

Some of the projects that use SQLite are catalogued at http://www.sqlite.org/cvstrac/wiki?p=SqliteUsers, and you are encouraged to add details of your own projects here too.

Because SQLite 3 has a new API and a new database file format, this book deals primarily with version 2 because it is already available, stable, and well supported. The new features of SQLite 3 are covered in Appendix I, "The Future of SQLite."

Features and Limitations

In this section we will look at the key features of SQLite and some of its limitations. The nature of SQLite makes it an ideal choice for quite a number of tasks, but it's not suitable for everything.

It is important to decide whether SQLiteor any other database engine for that matteris the right choice for your application before committing to a particular technology.

Speed

SQLite is extremely efficient, benefiting from a highly optimized internal architecture and a small memory footprint. Because SQLite is not a client/server database, the overheads of running a database daemon and socket communication are eliminated.

The published speed comparison at http://www.sqlite.org/speed.html compares SQLite to both MySQL and PostgreSQL. It finds that SQLite can perform up to 20 times faster than PostgreSQL and more than twice as fast as MySQL for common operations.

These tests were performed with default installations of each database, and although it is possible to tune the MySQL and PostgreSQL servers for slightly better performance in a given environment, SQLite does not require any such optimization.

The tests found that SQLite is significantly slower than the other databases only on the operations to create an index and to drop a table. However, slowness in these areas will not affect performance on a production database.

Portability

Because SQLite databases are stored as single files on the filesystem, they are very portable indeed. A database can be copied from one file to another, even across different operating systems. This means that for a cross-platform distribution you just need to concentrate on making your code portableeven when a populated database is to be shipped with the application.

SQLite has no external dependencies. The SQLite library is self-contained, so the only system requirement to run an application with an embedded SQLite database is the SQLite library itself. Because SQLite can be freely distributed, you can always ensure that this is present.

Security

SQLite databases are stored to the filesystem and access control is performed by the underlying operating system based on that file's permission settings.

Though SQLite can be accessed by processes running as different users if the correct file permissions are set, the database engine does not detect which user is performing a particular operation.

The advantage of this is one of administrative simplicitythere is no need to set up a complex user grants scheme. Any user who has access to read the database file is able to access the database tables and records. Likewise, in a shared environment, users are able to create their own SQLite databases to their file space without any involvement from the system administrators.

The disadvantage comes when you want to control permissions at a more finely grained level. There is no GRANT operation that would allow access to particular tables to one set of users but not others. If users have read access, they are able to read the entire database, and if they have write access, you have to be sure of their competence and trustworthiness with the data!

SQL Implementation

When Not to Choose SQLite

There is no hard-and-fast rule that determines whether SQLite is the right choice for your application. It is a robust, fast database engine that implements as much SQL as you are likely to need, but in some situations there are probably better choices.

In this section we will look at some circumstances that require features that SQLite cannot provide, followed by some examples of situations where SQLite is a very good choice indeed.

When SQLite Is Probably Wrong

First the bad news: The following situations may be better handled by another database system.

Network or Client/Server Applications

SQLite is not well suited for multiuser access over a network. SQLite can read from and write to its databases on a network share, but performance will take a hit because of the high latency levels that are usually found on a network filesystem.

File locking can often be patchy across a network, which could lead to two SQLite processes writing to the database at the same time, inevitably causing corruption.

A client/server RDBMS avoids these issues because all filesystem access is performed by the server after its requests have been received using a network protocol.

High-Volume Websites

For the vast majority of websites SQLite is a good choice. However, some sites receive so much traffic that their database components would be better suited to a client/server RDBMS.

How to quantify high traffic volume is the million-dollar question. The traffic level itself is not as much of an issue as the number of database reads and, more importantly, writes that are performed when a typical visitor does some surfing around.

Because web servers can send multiple requests simultaneously, if the database is frequently locked these processes will often be hanging around waiting for each other to finish writing before they can do their own thing.

The last thing your busy website needs is a database bottleneck, so you should consider a client/server RDBMS with more finely grained database locking.

High Concurrency

Similar to the reasons that a high-volume website may not be suitable for SQLite, a multiprocess or multithreaded application that performs a large number of database accesses may run into file-locking issues that could be avoided in an RDBMS that implements locking on smaller subsets of the database.

Again, quantifying a high level of concurrency is difficult, but this is something you should consider if your application will attempt to perform many concurrent database operations.

Each lock on the database file may only last for a few milliseconds, but there are still times where this will be too long for the application to work efficiently.

When SQLite Is Probably Right

Now for the good news: There are many applications for which SQLite is a great choice.

Looking at SQLite Databases

As we have seen, SQLite provides APIs to allow a database to be embedded into a wide variety of programming languages. However, there are times when you want to take a look inside your database without writing a program to do so. This section introduces two tools that can be used to query and amend SQLite database files.

The sqlite Tool

SQLite comes bundled with the sqlite tool, which provides a command-line interface to work with database files. It is invoked from the shell taking simply the name of a database as its argument.

```
$ sqlite dbfile
SQLite version 2.8.15
Enter ".help" for instructions
sqlite>
```

The sqlite> prompt indicates that sqlite is waiting for a new command to be entered. There are two types of commandseither an SQL statement terminated with a semicolon or a command relating to the sqlite program itself, beginning with a dot.

The .help command lists all the dot commands available along with a brief description, as shown in the following:

```
sqlite> .help
.databases              List names and files of attached databases
.dump ?TABLE? ...       Dump the database in a text format
.echo ON|OFF            Turn command echo on or off
.exit                   Exit this program
.explain ON|OFF         Turn output mode suitable for EXPLAIN on or off.
.header(s) ON|OFF       Turn display of headers on or off
.help                   Show this message
.indices TABLE          Show names of all indices on TABLE
.mode MODE              Set mode to one of "line(s)", "column(s)",
                        "insert", "list", or "html"
.mode insert TABLE      Generate SQL insert statements for TABLE
.nullvalue STRING       Print STRING instead of nothing for NULL data
.output FILENAME        Send output to FILENAME
.output stdout          Send output to the screen
.prompt MAIN CONTINUE   Replace the standard prompts
.quit                   Exit this program
.read FILENAME          Execute SQL in FILENAME
.schema ?TABLE?         Show the CREATE statements
.separator STRING       Change separator string for "list" mode
.show                   Show the current values for various settings
.tables ?PATTERN?       List names of tables matching a pattern
.timeout MS             Try opening locked tables for MS milliseconds
.width NUM NUM ...      Set column widths for "column" mode
```

Let's issue a few simple SQL commands to create a new table and add some rows of data. Don't worry too much about how these commands work for now; we just need to get something into the database to show how sqlite works.

First of all, the following statement creates a new table called mytable that has two columns, col1 and col2.

```
sqlite> CREATE TABLE mytable (
   ...>    col1 NUMERIC,
   ...>    col2 TEXT
   ...> );
sqlite>
```

Help and Support

Before we get into using SQLite in the following chapters, let's take a look at the support available to you if you run into problems.

There is some very good online documentation at http://www.sqlite.org/docs.html, which is updated regularly when new features are added to SQLite.

If you need something more interactive, the SQLite mailing list is a good place to ask questions. Send a blank email to sqlite-users-subscribe@sqlite.org to join the list, or if you'd rather receive a daily digest of messages, send it to sqlite-users-digest-subscribe@sqlite.org. The mailing list is frequented by the author and many regular users of SQLite, so you should find someone who is able to give good advice.

If professional support for your database engine is a requirement, this is available from Hipp, Wyrick & Company, Inc. (known as Hwaci for short) for a fee. Support is provided by the author and maintainer of SQLite, D. Richard Hipp, so you know you will be talking to an authority when your moment of need arises. More information can be found at http://www.hwaci.com/sw/sqlite/prosupport.html.

Chapter 2. Working with Data

We begin this chapter by getting our hands dirty right away with a basic tutorial that will give you an overview of how to work with data in SQLite.

SQLite Basics

If you have used SQL with other database systems, the language elements will be familiar to you, but it is still worth following the tutorial to see some of the differences between SQLite's implementation of SQL and the version you are used to.

If you are new to SQL, you will begin to pick up the basics of the language by following the examples, and we will go into more depth on each topic in the following chapters.

Prerequisites

To follow the examples in this tutorial you will need access to a workstation or space on a server system on which you can create a SQLite database.

You will also need the sqlite command-line tool to be installed and in your path. Full installation instructions can be found in Appendix A, "Downloading and Installing SQLite," but for now the quickest way to get started is to use one of the precompiled sqlite binaries from http://www.hwaci.com/sw/sqlite/download.html.

Remember, SQLite writes its databases to the filesystem and does not require a database server to be running. The single executable sqlite (or sqlite.exe on Windows systems) is all that you need.

Obtaining the Sample Database

All of the code in this book is available for download on the Sams Publishing website at http://www.samspublishing.com/. Enter this book's ISBN (without the hyphens) in the Search box and click Search. When the book's title is displayed, click the title to go to a page where you can download the code to save you from retyping all the CREATE TABLE commands that follow. However, you don't have to download the sample database as we have included the full set of commands required for the tutorial in this book.

Creating and Connecting to a Database

SQLite stores its databases in files, and you should specify the filename or path to the file when the sqlite utility is invoked. If the filename given does not exist, a new one is created; otherwise, you will connect to the database that you specified as the filename.

```
$ sqlite demodb
SQLite Version 2.8.12
Enter ".help" for instructions
sqlite>
```

Even without executing any SQLite commands, without any tables being created, connecting to a database as indicated in the preceding example will create an empty database file with the name specified on the command line.

```
$ ls l
-rw-r--r--    1 chris    chris           0 Apr  1 12:00 demodb
```

No file extension is required or assigned by sqlite; the database file is created with the exact name specified. So how do we know that this is a SQLite database? You could opt for a consistent file extension, for example using somename.db (as long as .db is a unique extension for your system) or use a well-organized directory structure to separate your data from your program files.

When there is something inside your database, you can identify the file as a SQLite database by taking a look at the first few bytes using a binary-safe paging program such as less. The first line you will see will look like this:

```
** This file contains an SQLite 2.1 database **
```

Querying and Updating the Database

So now that we have some records in our database, let's look at ways to fetch, modify, and delete records using SQLite's implementation of SQL.

The SELECT Statement

Use SELECT to fetch rows from a table. Either use a comma-separated list of column names or use an asterisk to fetch all columns in the table.

```
sqlite> SELECT first_name, email FROM employees;
Alex|alex@mycompany.com
Brenda|brenda@mycompany.com
Colin|colin@mycompany.com
Debbie|peter@mycompany.com

sqlite> SELECT * FROM clients;
501|Acme Products|Mr R. Runner|555-6800
502|ABC Enterprises|Mr T. Boss|555-2999
503|Premier Things Ltd|Mr U. First|555-4001
```

You'll notice that the format of the output isn't as readable as it could be. The sqlite program has a number of formatting options of which this mode, a pipe-separated list, is the default. The various output modes were discussed in Chapter 1, "Getting Started."

Although character-separated output can be a useful format for parsing by a program, a tabulated output is clearer in print. Therefore, for the rest of this tutorial we will use column mode with headings turned on.

```
sqlite> .mode column
sqlite> .header on
sqlite> select * from clients;
id          company_name   contact_name  telephone
----------  -------------  ------------  ----------
501         Acme Products  Mr R. Runner  555-6800
502         ABC Enterpris  Mr T. Boss    555-2999
503         Premier Thing  Mr U. First   555-4001
504         Integer Prima  Mr A. Increm  555-1234
```

The values of company_name and contact_name in the preceding code are truncated. Their full values are stored in the database, but the column display format causes the output to be arbitrarily limited. You can specify the width settings in sqlite with the .width command:

```
sqlite> .mode column
sqlite> .width 4 24 15 10
sqlite> select * from clients;
id    company_name              contact_name     telephone
----  ------------------------  ---------------  ----------
501   Acme Products             Mr R. Runner     555-6800
502   ABC Enterprises           Mr T. Boss       555-2999
503   Premier Thing             Mr U. First      555-4001
504   Integer Primary Key Ltd   Mr A. Increment  555-1234
```

For the rest of this tutorial, display widths have been adjusted to suit the output and the actual .width commands issued are not shown in the code.

The WHERE Clause

Chapter 3. SQLite Syntax and Use

In this chapter we look in detail at the SQL syntax understood by SQLite. We will discuss the full capabilities of the language and you will learn to write effective, accurate SQL.

You have already come across most of the supported SQL commands in Chapter 2, "Working with Data," in the context of the demo database. This chapter builds on that knowledge by exploring the syntax and usage of each command in more detail to give a very broad overview of what you can do using SQLite.

Naming Conventions

Each database, table, column, index, trigger, or view has a name by which it is identified and almost always the name is supplied by the developer. The rules governing how a valid identifier is formed in SQLite are set out in the next few sections.

Valid Characters

An identifier name must begin with a letter or the underscore character, which may be followed by a number of alphanumeric characters or underscores. No other characters may be present. These identifier names are valid:

- mytable
- my_field
- xyz123
- a

However, the following are not valid identifiers:

- my table
- my-field
- 123xyz

You can use other characters in identifiers if they are enclosed in double quotes (or square brackets), for example:

```
sqlite> CREATE TABLE "123 456"("hello-world", " ");
```

Name Length

SQLite does not have a fixed upper limit on the length of an identifier name, so any name that you find manageable to work with is suitable.

Reserved Keywords

Care must be taken when using SQLite keywords as identifier names. As a general rule of thumb you should try to avoid using any keywords from the SQL language as identifiers, although if you really want to do so, they can be used providing they are enclosed in square brackets.

For instance the following statement will work just fine, but this should not be mimicked on a real database for the sake of your own sanity.

```
sqlite> CREATE TABLE [TABLE] (
   ...>    [SELECT],
   ...>    [INTEGER] INTEGER
```

Creating and Dropping Tables

Creating and dropping database tables in SQLite is performed with the CREATE TABLE and DROP TABLE commands respectively. The basic syntax for CREATE TABLE is as follows:

```
CREATE [TEMP | TEMPORARY] TABLE table-name (
 column-def[, column-def]*
   [,constraint]*
);
```

Simply put, a table may be declared as temporary, if desired, and the structure of each table has to have one or more column definitions followed by zero or more constraints.

Table Column Definitions

A column definition is defined as follows:

```
name [type] [[CONSTRAINT name] column-constraint]*
```

As you saw in Chapter 2, SQLite is typeless and therefore the type attribute is actually optional. Except for an INTEGER PRIMARY KEY column, the data type is only used to determine whether values stored in that column are to be treated as strings or numbers when compared to other values.

You can use the optional CONSTRAINT clause to specify one or more of the following column constraints that should be enforced when data is inserted:

- NOT NULL
- DEFAULT
- PRIMARY KEY
- UNIQUE

A column declared as NOT NULL must contain a value; otherwise, an INSERT attempt will fail, as demonstrated in the following example:

```
sqlite> CREATE TABLE vegetables (
   ...>    name CHAR NOT NULL,
   ...>    color CHAR NOT NULL
   ...> );

sqlite> INSERT INTO vegetables (name) VALUES ('potato');
SQL error: vegetables.color may not be NULL
```

Often, a column declared NOT NULL is also given a DEFAULT value, which will be used automatically if that column is not specified in an INSERT. The following example shows this in action.

```
sqlite> CREATE TABLE vegetables (
   ...>    name CHAR NOT NULL,
```

Anatomy of a SELECT Statement

The syntax definition for an SQL statement is

```
SELECT [ALL | DISTINCT] result [FROM table-list]
[WHERE expr]
[GROUP BY expr-list]
[HAVING expr]
[compound-op select]*
[ORDER BY sort-expr-list]
[LIMIT integer [(OFFSET|,) integer]]
```

The only required item in a SELECT statement is the result, which can be one of the following:

- The * character
- A comma-separated list of one or more column names
- An expression

The latter two bullet points should be combined into: "A comma-separated list of one or more expressions." The original two points make it seem as if the following would be an error:

```
SELECT a+1, b+1 FROM ab;
```

but this would be okay:

```
SELECT a, b FROM ab;
```

In fact, both are valid.

Using the * character or a list of columns makes no sense without a FROM clause, but in fact an expression whose arguments are constants rather than database items can be used alone in a SELECT statement, as in the following examples.

```
sqlite> SELECT (60 * 60 * 24);
86400

sqlite> SELECT max(5, 20, -4, 8.7);
20

sqlite> SELECT random();
220860261
```

If the FROM list is omitted, SQLite effectively evaluates the expression against a table that always contains a single row.

The FROM list includes one or more table names in a comma-separated list, each with an optional alias name that can be used to qualify individual column names in the result. Where aliases are not used, the table name in full can be used to qualify columns.

Attaching to Another Database

Using sqlite, the .databases command lists all the databases that are open for the current session. There will always be two databases open after you invoke sqlitemain, the database specified on the command line, and temp, the database used for temporary tables.

```
sqlite> .databases
0     main        /home/chris/sqlite/demodb
1     temp        /var/tmp/sqlite VGazbfyWvuUr29P
```

It is possible to attach more databases to your current session with the ATTACH DATABASE statement. This adds a connection to another database without replacing your currently selected database.

The syntax is

```
ATTACH [DATABASE] database-filename AS database-name
```

The keyword DATABASE is optional and is used only for readability, but you must provide a unique database-name parameter that will be used to qualify table references, essential in case more than one database could have the same table name.

Suppose you are working on a new database called newdb and want to access some of the databases from our demo database from Chapter 2. The following example shows demodb being attached to the current sqlite session:

```
$ sqlite newdb
SQLite version 2.8.12
Enter ".help" for instructions
sqlite> ATTACH DATABASE demodb AS demodb;
sqlite> .databases
0     main        /home/chris/sqlite/newdb
1     temp        /var/tmp/sqlite_VGazbfyWvuUr29P
2     demodb      /home/chris/sqlite/demodb
```

Accessing tables from an attached database is straightforwardjust prefix any table name with the database name (the name given after the keyword AS, not the filename, if they are different) and a period.

We can perform a query on the clients table from demodb as follows:

```
sqlite> SELECT company_name FROM demodb.clients;
company_name
--------------------
Acme Products
ABC Enterprises
Premier Things Ltd
```

Tables in the main database can be accessed using their table name alone, or qualified as main.tablename. If a table name is unique across all databases attached in a particular session, it does not need to be prefixed with its database name even if it is not in the main database. However, it is still good practice to qualify all tables when you are working with multiple databases to avoid confusion.

Note

The SQL commands INSERT, UPDATE, SELECT, and DELETE can all be performed on an attached database by using the database name prefix. However, CREATE TABLE and DROP TABLE can only take place on the main databaseyou must exit sqlite and begin a new session if you want to manipulate tables from a different database.

Manipulating Data

Next we'll look at how records can be added to a database and demonstrate different ways of using the INSERT command, and examine the syntax of the SQL UPDATE and DELETE commands.

Transactions

Any change to a SQLite database must take place within a transactiona block of one or more statements that alter the database in some way. Transactions are the way in which a robust database system ensures that either all or none of the requests to alter the database is carried out; it can never be just partially completed. This property of a database is called atomicity.

Whenever an INSERT, UPDATE, or DELETE command is issued, SQLite will begin a new transaction unless one has already been started. An implicit transaction lasts only for the duration of the one statement but ensures that, for instance, an UPDATE affecting many rows of a large table will always carry out the action on every row orin the unlikely event of a system failure while processing this commandnone of them. The database will not reflect a change to any row until every row has been updated and the transaction closed.

A transaction can be started from SQL if you want to make a series of changes to the database as one atomic unit. This is the syntax of the BEGIN TRANSACTION statement:

```
BEGIN [TRANSACTION [name]] [ON CONFLICT conflict-algorithm]
```

The transaction name is optional and, currently, is ignored by SQLite. The facility to provide a transaction name is included for future use if the ability to nest transactions is added. Currently only one transaction can be open at a time. In fact the keyword trANSACTION is also optional, but is included for readability.

An ON CONFLICT clause can be specified to override the default conflict resolution algorithm specified at the table level, but can be superseded itself by the OR clause of an INSERT, UPDATE, or DELETE statement.

To end a transaction and save changes to the database, use COMMIT TRANSACTION. The optional transaction name may be specified. To abort a transaction without any of the changes being stored, use ROLLBACK TRANSACTION.

Inserting Data

There are two versions of the syntax for the INSERT statement, depending on where the data to be inserted is coming from.

The first syntax is the one we have already used in Chapter 2, to insert a single row from values provided in the statement itself. The second version is used to insert a dataset returned as the result of a SELECT statement.

INSERT Using VALUES

The syntax for a single-row insert using the VALUES keyword and a list of values provided as part of the statement is as follows:

```
INSERT [OR conflict-algorithm]
INTO [database-name .] table-name [(column-list)]
VALUES (value-list)
```

Although all our examples so far have included a column-list, it is actually optional. Where no column-list is provided, the value-list is assumed to contain one value for each column in the table, in the order they appear in the schema.

This can be a useful shortcut when you are adding data; for instance because we know the column-list is a name and

Indexes

The subject of keys and indexes and how they can affect the performance of your database will be addressed in Chapter 4, "Query Optimization," but first we will examine the syntax for creating and finding information on table indexes.

Creating and Dropping Indexes

The CREATE INDEX command is used to add a new index to a database table, using this syntax:

```
CREATE [UNIQUE] INDEX index-name
ON [database-name .] table-name (column-name [, column-name]*)
[ON CONFLICT conflict-algorithm]
```

The index-name is a user-provided identifier for the new index and must be unique across all database objects. It cannot take the same name as a table, view, or trigger. A popular naming convention is to use the table name and the column name(s) used for the index key separated by an underscore character.

To add an index to the color column of the vegetables table, we would use the following command.

```
sqlite> CREATE INDEX vegetables_color
   ...> ON vegetables(color);
```

The syntax of column-name allows for a sort order to be given after each column name, either ASC or DESC; however, currently in SQLite this is ignored. At the present time, all indexes are created in ascending order.

Removing an index is done with reference to the identifier given when it was created, which you can always find by querying the sqlite_master table if you cannot remember it.

```
sqlite> SELECT * FROM sqlite_master
  ...> WHERE type = 'index';
    type = index
    name = vegetables_color
tbl_name = vegetables
rootpage = 10
     sql = CREATE INDEX vegetables_color
ON vegetables(color)
```

The DROP INDEX command works as you might expect:

```
sqlite> DROP INDEX vegetables_color;
```

Don't worry if you misread the sqlite_master output and use the table name instead of the index name. SQLite only allows you to drop indexes with the DROP INDEX command and tables with the DROP TABLE command.

```
sqlite> DROP INDEX vegetables;
SQL error: no such index: vegetables
```

UNIQUE Indexes

The UNIQUE keyword is used to specify that every value in an indexed column is unique. Where an index is created on more than one column, every permutation of the column values has to be unique, even though the same value may appear more than once in its own column.

Views

A view is a convenient way of packaging a query into an object that can itself be used in the FROM clause of a SELECT statement.

Creating and Dropping Views

The syntax for CREATE VIEW is shown next.

```
CREATE [TEMP | TEMPORARY] VIEW view-name AS select-statement
```

The select-statement can be as simple or as complex as necessary; it could return the subset of a single table based on a conditional WHERE clause, or join many tables together to form a single object that can be more easily referenced in SQL.

To drop a view, simply use the DROP VIEW statement with the view-name given when it was created.

A view is not a table. You cannot perform an UPDATE, INSERT, COPY, or DELETE on a view, but if the data in one of the source tables changes, those changes are reflected instantly in the view.

Using Views

The following example shows a view based on the demo database tables employees and employee_rates using a query that returns the current rate of pay for each employee.

```
sqlite> CREATE VIEW current_pay AS
   ...> SELECT e.*, er.rate
   ...> FROM employees e, employee_rates er
   ...> WHERE e.id = er.employee_id
   ...> AND er.end_date IS NULL;
```

We can then query the new view directly, even adding a new condition in the process:

```
sqlite> SELECT * FROM current_pay
   ...> WHERE sex = 'M';
id    first_name  last_name   sex  email                      rate
----  ----------  ----------  ---  -------------------------  ------
101   Alex        Gladstone   M    alex@mycompany.com         30.00
103   Colin       Aynsley     M    colin@mycompany.com        25.00
```

The column names in a view are the column names from the table. Where an expression is used, SQLite will faithfully reproduce the expression as the column heading.

```
sqlite> CREATE VIEW veg_upper AS
   ...> SELECT upper(name), upper(color)
   ...> FROM vegetables;
sqlite> SELECT * FROM veg_upper LIMIT 1;
upper(name)|upper(color)
CARROT|GREEN
```

However, the column in the view cannot actually be called upper(name). As shown in the following example, SQLite will attempt to evaluate the upper() function on the nonexistent name column.

```
sqlite> SELECT upper(name) from veg_upper;
```

Triggers

A trigger is an event-driven rule on a database, where an operation is initiated when some other transaction (event) takes place. Triggers may be set to fire on any DELETE, INSERT, or UPDATE on a particular table, or on an UPDATE OF particular columns within a table.

Creating and Dropping Triggers

The syntax to create a trigger on a table is as follows:

```
CREATE [TEMP | TEMPORARY] TRIGGER trigger-name
[BEFORE | AFTER] database-event ON [database-name .]table-name
trigger-action
```

The TRigger-name is user-specified and must be unique across all objects in the databaseit cannot share the same name as a table, view, or index.

The trigger can be set to fire either BEFORE or AFTER database-event; that is, either to pre-empt the transaction and perform its action just before the UPDATE, INSERT, or DELETE takes place, or to wait until the operation has completed and then immediately carry out the required action.

If the database-event is specified as UPDATE OF column-list, it will create a trigger that will fire only when particular columns are affected. The trigger will ignore changes that do not affect one of the listed columns.

The TRigger-action is further defined as

```
[FOR EACH ROW | FOR EACH STATEMENT] [WHEN expression]
BEGIN
 trigger-step; [trigger-step;] *
END
```

At present only FOR EACH ROW TRiggers are supported, so each trigger stepwhich may be an INSERT, UPDATE, or DELETE statement or SELECT with a function expressionis performed once for every affected row in the transaction that causes the trigger to fire. The WHEN clause can be used to cause a trigger to fire only for rows for which the WHEN clause is true. The WHEN clause is formed in the same way as the WHERE clause in a SELECT statement.

The WHEN clause and any trigger-steps may reference elements of the affected row, both before and after the trigger action is carried out, as OLD.column-name and NEW.column-name respectively. For an UPDATE action both OLD and NEW are valid. An INSERT event can only provide a reference to the NEW value, whereas only OLD is valid for a DELETE event.

An ON CONFLICT clause can be specified in a trigger-step; however, any conflict resolution algorithm specified in the statement that causes the trigger to fire will override it.

As you might expect, the syntax to drop a trigger is simply

```
DROP TRIGGER [database-name .] table-name
```

If you forget the name of a trigger, you can query sqlite_master using type = 'trigger' to find all the triggers on the current database.

Using Triggers

In the last chapter we mentioned that triggers could be used to implement a cascading delete, so that rows from a

Working with Dates and Times

In our sample database we have chosen to use integers for columns that store a date value, represented by the format YYYYMMDD. This format is fairly readable and, because the most significant part (the year) comes first, allows arithmetic comparisons to be performed. For instance just as February 29th 2004 is earlier than March 1st, 20040229 is a smaller number than 20040301.

This technique is not without its limitations. First, there is no validation on the values stored. Although February 29th is a valid date in the leap year 2004, it does not exist three years out of four and the value 20050229 is not a real date, yet could still be stored in the integer column or compared to a real date.

In fact even if you used a trigger to make the number eight digits long and also fall within a sensible year range, there are many values that could still be stored that do not represent dates on the calendar. Very strict checking would be required in your application program to ensure such date information was valid.

Similarly, you cannot perform date arithmetic using integer dates. Although 20040101 + 7 gives a date seven days later, 20040330 + 7 would give a number that looks like March 37th.

We have not even looked at a data type to store a time value yet, but the same limitations apply if a numeric field is used. SQLite contains a number of functions that allow you to work with both dates and times stored as character strings, allowing you to manipulate the values in useful ways.

Valid Timestring Formats

SQLite is fairly flexible about the format in which you can specify a date and/or time. The valid time string formats are shown in the following list:

- YYYY-MM-DD
- YYYY-MM-DD HH:MM
- YYYY-MM-DD HH:MM:SS
- YYYY-MM-DD HH:MM:SS.SSS
- HH:MM
- HH:MM:SS
- HH:MM:SS.SSS
- now
- DDDD.DDDD

For the format strings that only specify a time, the date is assumed to be 2000-01-01. Where no time is specified,

SQL92 Features Not Supported

We finish this chapter on SQLite's implementation of the SQL language by looking at features of the ANSI SQL92 standard that are not currently supported by SQLite.

- Although the CREATE TABLE syntax permits an optional CHECK clause to be present, the CHECK constraint is not enforced.
- The keywords FOREIGN KEY are allowable in a CREATE TABLE statement; however, this currently has no effect.
- Subqueries must return a static data set, and they may not refer to variables in the outer queryalso known as correlated subqueries.
- All triggers are currently FOR EACH ROW, even if FOR EACH STATEMENT is specified.
- Views are read-only, even when they select only from one table. However, an INSTEAD OF trigger can fire on an attempted INSERT, UPDATE, or DELETE to a view and deal with the transaction in the desired manner.
- INSTEAD OF triggers are allowed only on views, not on tables.
- Recursive triggerstriggers that trigger themselvesare not supported.
- The ALTER TABLE statement is not present; instead a table must be dropped and re-created with the new schema.
- Transactions cannot be nested.
- count(DISTINCT column-name) cannot be used. However, this can be achieved by selecting a count() from a subselect of the desired table that uses the DISTINCT keyword.
- All outer joins must be written as LEFT OUTER JOIN. RIGHT OUTER JOIN and FULL OUTER JOIN are not recognized.
- The GRANT and REVOKE commands are meaningless in SQLitethe only permissions applicable are those on the database file itself.

Chapter 4. Query Optimization

Now that you are familiar with the SQL syntax implemented by SQLite, let's take a look at the way in which your tables are created and how the way your queries are written can affect the performance of your database.

Keys and Indexes

In the preceding chapter you met the CREATE INDEX statement, used to add an index to one or more columns of a database table.

Indexes are used to preserve a particular sort order on a column within the database itself, enabling queries in which the WHERE clause references that column directly to operate much faster. Rather than scanning a table's data from top to bottom and pulling out just the rows that match the given criteria, the index tells SQLite exactly where to look in that table to find the matching rows.

What an Index Does

A database index is not an abstract concept; you certainly have come across one before. Let's consider the index in this book, for instance, and think about when you would use it.

If you were trying to find references to the API call sqlite_exec(), for example, you might already expect to find this in Chapter 6, "The C/C++ Interface"but bear in mind that a computer program cannot think for itself. So for a moment pretend you don't have the benefit of this knowledge.

To make sure you find every reference in the book, the systematic and thorough approach is to start from page one, line one and read each page a line at a time until the end of the book, noting each reference as you come across it. Clearly, this would be a time-consuming process.

Fortunately the production team behind this book has created a comprehensive index, with topics and terms used throughout the book listed alphabetically and page references given for each entry. You've seen an index before and know how to use one, so you already know that it's much quicker to look up sqlite_exec() in the index and jump to each page in turn to find the topic you are looking for than to scan every single printed word.

How Indexes Work in SQLite

A database index works much the same way as the index in a book. Suppose you have a table containing a list of names and telephone numbers, and the data has been built up over time as you come into contact with new people. It's very likely that some of your acquaintances will share a surname, but certainly your records were not added to the table in alphabetical order.

Imagine you want to find the details for someone called Brown. The SQL SELECT statement might look like this:

```
SELECT * FROM contacts
WHERE last_name = 'Brown';
```

Without an index, the database has to look at each record in the database in turn and compare it to the string Brown. The more popular you are, the larger your contacts database will be and the longer this query will take to complete. This process is known as a full table scan.

However, having an index on the last_name column means that SQLite will know how to sort the records in that table alphabetically by surname, and consequently it can pick out the matching values of last_name without even looking at the other rows in the table.

Such an index could be created on the contacts table as follows:

```
CREATE INDEX contacts_last_name ON contacts(last_name);
```

Remember that each index has to be given its own unique identifier, and that an index name cannot share the same name as a table, view, or trigger in the same database.

The EXPLAIN Statement

There is a way to compare how different indexes affect the speed of certain queries without having to perform endless benchmark tests. However, it requires a little knowledge of the inner workings of SQLite.

The EXPLAIN command can be used to find out how an SQL query is parsed by SQLite and from that you can determine the way in which it will actually be executed. The output from EXPLAIN is a series of opcodes from the Virtual Database Engine, which we'll look at in more detail in Chapter 10, "General Database Administration."

When using sqlite the .explain command can be used to make the format of the output of EXPLAIN more readable. The following example shows the opcodes used to process a query on t2 using a WHERE condition on the non-indexed num column.

```
sqlite> .explain
sqlite> EXPLAIN SELECT word FROM t2 WHERE num = 1234;
addr  opcode        p1          p2          p3
----  ------------  ----------  ----------  ----------------------------------
0     ColumnName    0           0           word
1     Integer       0           0
2     OpenRead      0           4           t2
3     VerifyCookie  0           2979
4     Rewind        0           11
5     Column        0           0
6     Integer       1234        0           1234
7     Ne            1           10
8     Column        0           1
9     Callback      1           0
10    Next          0           5
11    Close         0           0
12    Halt          0           0
```

To work out whether an index is used or not, knowing the actual meaning of all these opcodes is not necessary. If you compare the preceding output to that for a similar query on t3 where num is indexed, you'll see the name of the index in the p3 column alongside an OpenRead opcode.

```
sqlite> EXPLAIN SELECT word FROM t3 WHERE num = 1234;
addr  opcode        p1          p2          p3
----  ------------  ----------  ----------  ----------------------------------
0     ColumnName    0           0           word
1     Integer       0           0
2     OpenRead      0           5           t3
3     VerifyCookie  0           2979
4     Integer       0           0
5     OpenRead      1           1077        t3_num_idx
6     Integer       1234        0           1234
7     NotNull       -1          10
8     Pop           1           0
9     Goto          0           22
10    MakeKey       1           0           n
11    MemStore      0           0
12    MoveTo        1           22
13    MemLoad       0           0
14    IdxGT         1           22
15    RowKey        1           0
16    IdxIsNull     1           21
17    IdxRecno      1           0
18    MoveTo        0           0
19    Column        0           1
20    Callback      1           0
21    Next          1           13
22    Close         0           0
23    Close         1           0
```

Part II: Using SQLite Programming Interfaces

Chapter 5. The PHP Interface

PHP is a server-side embedded scripting language for writing dynamic web pages that supports communication with many different databases, and SQLite is no exception.

In this chapter we will look at how SQLite support is activated in PHP, how to communicate with a database through a web page, and how custom functions can be added to SQLite through the PHP interface.

Configuring PHP for SQLite Support

Modern versions of PHP make it very easy to add in SQLite support, so we'll briefly look at the steps required for both Linux/Unix and Windows versions of PHP.

For more general assistance with PHP configuration on a particular platform, refer to the online documentation at http://www.php.net/manual/en/installation.php.

Configuring PHP for Linux/Unix

From PHP 5, the SQLite extension is bundled with the PHP distribution and will be enabled at compile time unless it is disabled with the --without-sqlite option.

You do not need to install any SQLite components before building PHP, but you will need to download the sqlite tool separately if you want to use it to examine your databases outside of PHP.

For earlier PHP versions, the official extension can be obtained from http://pecl.php.net/package/SQLite. This also includes source for libsqlite, so again there are no other prerequisites for adding the module to PHP. With PHP 4.3.0 and later, you can use the pear utility to automatically download and install the SQLite extension:

```
# pear install sqlite
downloading SQLite-1.0.2.tgz ...
Starting to download SQLite-1.0.2.tgz (362,412 bytes)
.....................................done: 362,412 bytes
51 source files, building
running: phpize
[...]
install ok: SQLite 1.0.2
```

The full output is quite lengthy, showing that the latest version of the extension has been downloaded, compiled, and copied to your PHP extensions directory.

For earlier versions of PHP, you can download and compile the extension by hand after downloading the latest version by following these steps:

```
# gzip d SQLite-1.0.2.tar.gz
# tar xf SQLite-1.0.2.tgz
# cd SQLite-1.0.2
# phpize
# ./configure
# make
# make install
```

With the extension installed, all you need to do to activate the new extension is add the following line to your php.ini file and restart the web server.

```
[sqlite]
extension="sqlite.so"
```

Alternatively you can also load the SQLite extension dynamically in each script that requires it with the dl() function.

```
dl("sqlite.so");
```

There should be no need to specify the full path to sqlite.so; the shared object file will be installed to the extension_dir value in php.ini.

Using the PHP SQLite Extension

Now we know that our PHP supports the SQLite interface, let's look at the set of commands that are available for communicating with a SQLite database.

Opening a Database

The PHP function to open a SQLite database is sqlite_open() and its prototype is given next:

```
resource sqlite_open ( string filename [, int mode [, string &errmessage]])
```

The filename parameter specifies the database name, and if the file specified does not exist it will be created. A relative or absolute path to the database file can be provided; otherwise, PHP looks for filename in the same directory as the script being executed. To open an in-memory database, use :memory:.

Although PHP can execute scripts from the command line, this chapter assumes that you are mostly using PHP in a web environment, where file permissions can be problematic. In order for PHP to open or to select from a SQLite database, the user under which the web server is running must have read permissions on the database file.

To perform an UPDATE, INSERT, or DELETE, that web server userid must also not only have write permissions on the database file itself but also must be able to create files within the same directory so that the journal file can be written.

Apache often runs as the user nobody or apache and so care needs to be takenparticularly on a shared systemif the database file is made writable to this user or, worse, if the file is made world-writable.

To prevent other users of the same web server from accessing your databases, PHP should be run in safe mode. An alternative is to use suPHPavailable from http://www.suphp.org/so that PHP can run as a number of different users on the same server.

The mode parameter to sqlite_open() specifies the mode of the file, and is octal 0666 by default. The intended use of this parameter is to open a database file in read-only mode, using mode 0444 for example.

The resource returned is a database handle if sqlite_open() is successful. On failure, the function returns FALSE and if the optional errmessage parameter is passed by reference it will contain a descriptive error message explaining why the database cannot be opened.

In Listing 5.2 we use sqlite_open() to open a new database webdb. To follow the rest of the examples in this chapter, make sure this database is created with the correct file permissions to be opened by PHP without any error.

Listing 5.2. Script to Open a Database with Error Checking

```
<?php

if (!$db = sqlite_open("webdb", 0666, &$errmessage)) {
  echo "Could not open database:<br />\n";
  echo $errmessage;
  exit;
}
else {
  echo "SQLite database opened";
}

?>
```

PHP allows for a persistent database connection to be used via the sqlite_popen() function, which takes the same

Working with User-Defined Functions

A very powerful feature of the PHP SQLite extension is the ability to define functions in PHP that can be registered and executed within an SQL statement. It is therefore possible for PHP developers to extend the functionality of SQLite using a familiar language.

The example in Listing 5.9 defines a trivial function in PHP that reverses each word in a string. This differs from the usual behavior of strrev(), which reverses the entire string. The example uses sqlite_create_function() to register a user-defined function, word_reverse(), for use within SQLite.

Listing 5.9. Adding a User-Defined Function to SQLite in PHP

```
<?php

function reverse_words($string) {
  $w = explode(" ", $string);
  for($i=0; $i<count($w); $i++) {
    if ($i > 0)
      $ret .= " ";
    $ret .= strrev($w[$i]);
  }
  return($ret);
}

if (!$db = sqlite_open("webdb", 0666, &$errmessage)) {
  echo "Could not open database:<br />\n";
  echo $errmessage;
  exit;
}
else {
  sqlite_create_function($db, "word_reverse", "reverse_words", 1);

  $sql = "SELECT word_reverse('The quick brown fox');";
  $res = sqlite_query($db, $sql);
  echo sqlite_column($res, 0);
}

?>P
```

First, we declare the function reverse_words() in PHP. Using explode() to break up the passed-in stringassuming words are separated by a space characterwe then call strrev() on each word in turn and piece the string back together:

```
function reverse_words($string) {
  $w = explode(" ", $string);
  for($i=0; $i<count($w); $i++) {
    if ($i > 0)
      $ret .= " ";
    $ret .= strrev($w[$i]);
  }
  return($ret);
}
```

After opening a SQLite connection to webdb as database resource $db, we use sqlite_create_function() to register the user-defined function.

```
sqlite_create_function($db, "word_reverse", "reverse_words", 1);
```

There are four arguments to sqlite_create_function(). First the database resource, in this case $db, is supplied. The

Using the PEAR Database Class

In addition to the API we have already discussed, PHP also allows access to SQLite databases via the PEAR Database class. We will show briefly how to use this with SQLite.

The PEAR class provides a unified API for accessing SQL-based databases. It supports SQLite as well as many other popular databases such as Microsoft SQL Server, MySQL, Oracle, and PostgreSQL, as well as ODBC connectivity.

The database-abstraction approach allows you to create scripts that can work on a wide range of database engines simply by changing the connection parameters. The same set of functions is made to work with whatever database type you happen to be connected to at the timeright down to a function that delimits quotes and other special characters the correct way for that database. If you expect your PHP application to need porting to a different database system in future, the PEAR class is well worth considering.

Note

Database-specific features must be worked around when using a database abstraction layer. For instance the INTEGER PRIMARY KEY feature specific to SQLite is similar to an AUTO INCREMENT field in MySQL, but the same functionality can be only achieved in Oracle using CREATE SEQUENCE. In fact the PEAR Database class forces you to use a database-level sequence object for all autoincrementing fields for maximum compatibility.

To download and install the PEAR DB class automatically if you do not already have it, use the pear utility:

```
# pear install DB
```

There are two ways to define a database connection. The first uses a string to indicate the Data Source Name (DSN) and usually looks like this:

```
dbtype://username:password@hostname/dbname
```

For SQLite, however, the username, password, and hostname fields are not required, so the following DSNs are sufficient to identify a database file in the current directory or in the given path respectively.

```
sqlite:///dbfile
sqlite:////home/chris/sqlite/dbfile
```

The second way to define a DSN is by setting elements of an array of parameters, which is defined by the class as follows:

```
$dsn = array(
    'phptype'  => false,
    'dbsyntax' => false,
    'username' => false,
    'password' => false,
    'protocol' => false,
    'hostspec' => false,
    'port'     => false,
    'socket'   => false,
    'database' => false,
);
```

Only the phptype and database elements are required to identify a SQLite database, so the following array is a valid DSN:

Chapter 6. The C/C++ Interface

In this chapter we will look at how SQLite's C/C++ interface can be incorporated into your programs. We will discuss the different ways of querying a database and look at how you can add your own custom functions to SQLite.

Preparing to Use the C/C++ Interface

In order to develop software with an embedded SQLite database, you need to make sure your system has the SQLite development library installed. The latest version can always be downloaded from http://www.sqlite.org/download.html.

If you have already installed and used sqlite as a precompiled binary, you will not necessarily have the library yet. For Linux, the precompiled library is sqlite.so.gz and for Windows it is sqlitedll.zip. If you also want to use the TCL bindings, precompiled libraries are available that support this, named tclsqlite.so.gz and tclsqlite.zip respectively.

Users of Red Hat Linux and other Unix-like systems that use the RPM package format can use the sqlite-devel package to install the library along with header files and documentation.

If you compiled SQLite from source, the libraries were installed when you issued the make install command. A default source installation on Linux/Unix will put libsqlite in the /usr/local/lib directory and sqlite.h in /usr/local/include.

Using the C Language Interface

Now that you have made sure the library is available on your system, let's take a look at the functions that make up the C language interface.

Opening and Closing a Database

The function to open a SQLite database is sqlite_open(). The prototype is

```
sqlite *sqlite_open(const char *dbname, int mode, char **errmsg);
```

The dbname parameter is the name of the database file on the filesystem. If just a filename is given, it is assumed that the database is in the current directory at runtime. A relative or absolute path can be specified.

The mode parameter is intended for future use, to specify the file mode in which the database file is opened. Typical values would be 0777 for read/write access or 0444 for read-only. However, at the time of writing, this parameter is ignored by the library.

The sqlite structure to which a pointer is returned is an opaque structure that must be passed as the first parameter to all the other SQLite API functions. If the database cannot be opened, the return value will be NULL and errmsg will point to the location of the error message created by sqlite_open(). The memory allocated for errmsg should be freed using sqlite_freemem().

To close the database, use sqlite_close(), which is simply passed the open sqlite structure pointer.

The example in Listing 6.1 shows how to establish a connection to a database called progdb in the current directory, and displays an error message if the database cannot be opened.

Listing 6.1. Connecting to a SQLite Database Using C/C++

```
#include <stdio.h>
#include <sqlite.h>

main()
{

  char *errmsg;

  sqlite *db = sqlite_open("progdb", 0777, &errmsg);

  if (db == 0)
  {
    fprintf(stderr, "Could not open database: %s\n", errmsg);
    sqlite_freemem(errmsg);
    exit(1);
  }

  printf("Successfully connected to database\n");
  sqlite_close(db);
}
```

We compile listing6.1.c into an executable listing6.1 as follows:

```
$ gcc -o listing6.1 listing6.1.c lsqlite
```

Executing the new program should tell us that we have been able to connect to the progdb database. The first time you run the program, the database will be created and you will be able to see a file called progdb in the current

Adding New SQL Functions

The SQLite library allows the SQL language to be extended with new functions implemented as C code. In fact all of SQLite's built-in functions are now written this way.

Creating a regular function begins with a call to sqlite_create_function(), and this is the prototype:

```
int sqlite_create_function() {
  sqlite *db,
  const char *zName,
  int nArg,
  void (*xFunc)(sqlite_func *, int, const char **),
  void *pUserData
);
```

The example in Listing 6.8 defines a trivial function in C that capitalizes every alternate letter in a string. The example uses sqlite_create_function() to register the function defined in capitalize_alternate() for use within SQLite under the name altcaps().

Listing 6.8. Creating a Custom Function Using sqlite_create_function()

```
#include <stdio.h>
#include <sqlite.h>


void capitalize_alternate(sqlite_func *context, int argc, const char **argv)
{
  int i;
  static char str[80];

  for (i=0; i<strlen(argv[0]); i++) {
    if (i%2 == 0)
      str[i] = toupper(argv[0][i]);
    else
      str[i] = tolower(argv[0][i]);
  }
  str[i] = '\0';

  sqlite_set_result_string(context, str, -1);
}

main()
{

  char *errmsg;
  char **result;
  char str[80];
  int ret, rows, cols, i, j;

  sqlite *db = sqlite_open("progdb", 0777, &errmsg);

  if (db == 0)
  {
    fprintf(stderr, "Could not open database: %s\n", errmsg);
    sqlite_freemem(errmsg);
    exit(1);
  }

  sqlite_create_function(db, "altcaps", 1, capitalize_alternate, NULL);

  ret = sqlite_get_table(db, "SELECT altcaps('this is a test')",
                                              &result, &rows, &cols, &errmsg);
```

Chapter 7. The Perl Interface

In this chapter we will look at the Perl interface to SQLite. Databases are accessed from Perl scripts using the Database Interface (DBI) and a Database Driver (DBD) for SQLite. If you have used DBI/DBD to interface Perl with other database systems, much of the procedure to communicate with SQLite will be familiar to you.

This chapter gives an overview of Perl's DBI in general and also details the attributes and methods specific to the SQLite DBD.

Preparing to Use the SQLite Interface

To access a SQLite database from Perl you need to install both the DBI module and the DBD module for SQLite.

If you do not already have DBI installed, use cpan to download it from the Comprehensive Perl Archive Network and install it.

```
# cpan
cpan> install DBI
```

On Windows platforms using the ActivePerl distribution, use the ppm.bat script to install Perl modules.

```
C:\perl\bin> ppm.bat
ppm> install DBI
```

Note that it can take a while to download, configure, compile, and install CPAN modules. If you are accessing CPAN for the first time, a set of modules will also be downloaded first to update your system.

To add the SQLite DBD module to your system, install DBD::SQLite2 from CPAN.

```
# cpan
cpan> install DBD::SQLite2
```

The DBD::SQLite2 package includes as much of SQLite as it needs, so there is no need to have the SQLite libraries on your system before installing the DBD moduleyou can add SQLite support to Perl on a fresh system without needing to download anything from sqlite.org, although you will probably want to install the sqlite monitor tool too.

Note

As we have chosen to stick with the existing, stable, and well-supported SQLite 2 engine throughout this book, you should install DBD::SQLite2 for a compatible Perl database driver. The original DBD::SQLite module uses the latest SQLite libraryversion 3which uses a different database file format than the previous version.

Though the Perl DBI abstracts the actual database back end and the examples in this chapter will work with whatever SQLite version you use, you also need to be using the appropriate sqlite tool for your SQLite library version in order to read your databases. You can read about the changes in SQLite 3 in Appendix I, "The Future of SQLite."

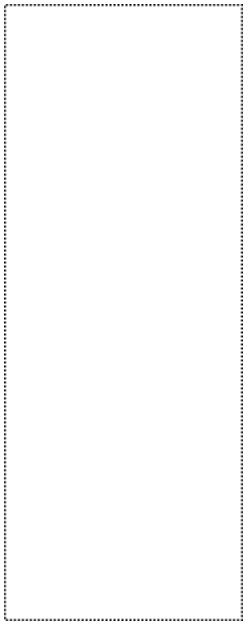
About the Perl DBI

Perl's Database Interface uses a database abstraction model, so it doesn't really matter what the underlying database is. The DBI module calls the appropriate Database Driver module and passes off the SQL commands for execution.

There are many more DBD modules than you are ever likely to use, supporting all the major database systems and many of the minor ones. The naming convention is DBD::dbname, for example DBD::Oracle for Oracle or DBD::Sybase for Sybase (and SQL Server). There's also DBD::ODBC for other ODBC-enabled databases that don't have their own specific DBD module and even DBD::CSV to interact with comma-separated-values files using DBI.

The DBD module we are interested in is called DBD::SQLite2, and the way in which queries from your Perl script interact with DBI and the SQLite DBD driver is shown in Figure 7.1.

Figure 7.1. The Perl Database Interface model.



The DBI module is loaded into your script with this simple command:

```
use DBI;
```

You do not need to use any specific DBD module as DBI will take care of that when it is needed. This makes it easy to write highly portable database applications in Perl because only a single instruction needs to be changed to tell DBI to work with a different database. The instruction looks like this for SQLite:

```
$dbh = DBI->connect("DBI:SQLite2:dbname=dbfile", "", "");
```

The prototype for DBI->connect() has three parametersa data source, username, and password. As SQLite does not use user-based authentication, the second and third parameters are always blank.

The data source contains three parts separated by colons: the keyword DBI, the database typein this case SQLiteand an expression indicating the name of the database. The filename given as dbfile can include an absolute or relative path or will be opened from the current working directory if no path is given.

Getting Information About the DBI

Using the SQLite DBD

In this section we will look at the methods and attributes available for a DBI object with examples specific to the DBD::SQLite2 module.

Opening and Closing the Database

As we saw before, the DBI->connect() function is used to open a database, and for a SQLite database no username or password arguments are required. The usage is always as follows:

```
$dbh = DBI->connect("DBI:SQLite2:dbname=dbfile", "", "");
```

Some basic error trapping is useful in case the connection to the database fails. Because SQLite will create a new database file with the given name if one does not exist, DBI->connect() will only fail if there is an I/O error, for instance file permissions not allowing the file to be opened or created in a particular directory, or no more disk space on the device.

The errstr property contains the most recent database error. The following example could be used to exit with an error message if SQLite is unable to open the specified dbfile:

```
$dbh = DBI->connect("DBI:SQLite2:dbname=perldb", "", "")
       or die("Error: " . DBI::errstr);
```

To close the connection to a database, simply invoke the disconnect() method on your DBI object.

```
$dbh->disconnect();
```

Executing SQL Statements

We have already seen that the do() method can be used to pass SQL to the DBI module in order to execute a command. In fact, do() is good only for non-SELECT statements and the preferred method to send SQL commands to the DBI module is using a two-step process. do() is a convenience function that effectively combines these two steps.

First the query needs to be prepared by the SQL engine using prepare, after which it can be executed using the execute method.

Listing 7.3 creates the contacts table in SQLite with error trapping at every stage. The errstr property returns the most recent error message generated by the DBD, so any cause of error that causes the script to exit prematurely will be displayed to the screen.

Listing 7.3. Creating a Table Using DBI

```
#!/usr/bin/perl -w

use strict;
use DBI;

my $dbh = DBI->connect("DBI:SQLite2:dbname=perldb", "", "")
          or die("Cannot connect: " . DBI::errstr() );

my $sql = "CREATE TABLE contacts (   ".
          "  id INTEGER PRIMARY KEY, ".
          "  first_name TEXT,        ".
          "  last_name TEXT,         ".
          "  email TEXT)             ";
```

Adding New SQL Functions

A powerful feature of the SQLite library is the ability to add user-defined functions to the SQL language. Because this feature is specific to SQLite, it is not part of the Perl DBI.

Instead, Perl provides the facility to create user-defined functions through two private methods in the SQLite DBD.

Creating Functions

You can register a new function using the private method create_function, called as follows:

```
$dbh->func( $name, $argc, $func_ref, "create_function" )
```

The three arguments required are the function name, the number of arguments the function will take, and a reference to the Perl function that is to be used whenever the SQL function is called.

The simplest way to see how a user-defined function is used is to register a built-in Perl function in the SQL language. The following statement uses a minimal inline function, such as $func_ref, which registers the Perl function rand() as the SQL function rand(). No arguments are required for the function, so $argc is zero.

```
$dbh->func( "rand", 0, sub { return rand() }, "create_function" );
```

SQLite's built-in random() function returns a random signed 32-bit integer, whereas Perl's rand() function returns a decimal between 0 and 1. You could see the difference by preparing and executing the following statement from a Perl script after the rand() function has been registered in SQL:

```
SELECT rand(), random();
```

User-defined functions have to be registered for each database connection where they are to be used. Such functions are available in SQL only for the duration of that connectionthey are not saved to the database itself or made available to other connection objects within the same script. Of course, user-defined functions are designed to allow you to add custom functions to SQL, so let's look at a more complex example (see Listing 7.11).

Listing 7.11. Creating a User-Defined Function in Perl

```
#!/usr/bin/perl -w

sub altcaps {
  my $str = $_[0];
  my $newstr = "";
  for ($i=0; $i<length($str); $i++) {
    $char = substr($str, $i, 1);
    if ($i%2) {
      $newstr .= uc($char);
    }
    else {
      $newstr .= lc($char);
    }
  }
  return $newstr;
}

use strict;
use DBI;

my $dbh = DBI->connect("DBI:SQLite2:dbname=perldb", "", "")
       or die("Cannot connect: " . DBI::errstr() );
```

Chapter 8. The Tcl Interface

Tcl (Tool Command Language) is a flexible, portable scripting language interpreter that is ideal for rapid application prototyping. With its simple syntax and in conjunction with Tk, a graphical user interface toolkit shipped as standard with Tcl, Tcl makes it possible to create powerful GUI applications very quickly.

In this chapter we look at how to add database functionality to your application using SQLite's Tcl interface.

Preparing to Use the Tcl Interface

The Tcl interface is included with the SQLite source distribution. Compiling from source on a Unix system creates the library file libtclsqlite.so, which should be installed to a valid library location.

A binary distribution of the TCL library can be obtained from http://www.sqlite.org/download.html for both Linux and Windows, in compressed formats named tclsqlite.so.gz and tclsqlite.zip respectively.

The library file should go in a location from which Tcl can find the package. On Linux this would usually be a subdirectory of /usr/share/tcl; on Windows, sqlite.dll is typically extracted to C:\tcl\lib.

Assuming the subdirectory is called TclSqlite, running the following command from Tcl will generate pkgIndex.tclwhich is required in order for Tcl to be able to locate the packageon a Linux system:

```
% pkg_mkIndex -direct /usr/share/tcl/TclSqlite *.so
```

On Windows, the equivalent command would be

```
% pkg_mkIndex -direct C:\tcl\lib\TclSqlite *.dll
```

If you installed using the RPM distribution on a compatible Linux system, tclsqlite.so will already have been installed to a directory from which it can be imported into tclsh or wish.

Using the Tcl Interface

Now that you have the SQLite Tcl library installed, let's look at how the interface is used within a Tcl script. In this section you'll see how to open and close a database and to issue commands that insert, update, or delete rows, and query the database. You'll also learn how to handle the resulting dataset in your script.

Opening and Closing a Database

To begin using the SQLite library from Tcl, your scripts first have to import the package.

```
package require sqlite
```

Run interactively, tclsh will display the version number of the SQLite library.

```
$ tclsh
% package require sqlite
2.0
```

You can access SQLite via Tcl by using a single command named sqlite, which you call as follows:

```
sqlite dbcmd database-name
```

The database file given in the database-name argument is openedor created if it does not exist. SQLite will attempt to open database-name from the current directory unless a path is specified.

A new composite command with the name given for dbcmd is then created for use within Tcl. The new command interfaces with an open SQLite database using a number of methods, similar to the way in which Tk widgets are created.

The first method we'll look at is the simplestthe close method instructs SQLite to close an open database, after which it will destroy the dbcmd command.

The first Tcl command in the following example would open a database called tcldb from the current directory and bind it to the command db1 within Tcl. The second command then closes the database, after which the command db1 will no longer be available.

```
$ tclsh
% package require sqlite
2.0
% sqlite db1 tcldb
0x95c7eb0
% db1 close
```

As usual, the error messages Tcl displays are helpful and generally no further error-trapping code is required. For instance, if you had tried to execute the preceding code in a directory that you did not have permission to write to, the error displayed would look like this:

```
$ tclsh
% package require sqlite
2.0
% sqlite db1 tcldb
unable to open database: tcldb
```

Chapter 9. The Python Interface

Python is an interpreted object-oriented programming language that is portable across different platforms. Its popularity is due to a combination of its powerful features and the clarity and agility of its syntax. It is an ideal language for prototyping while still managing to keep the code readable and manageable.

You can access SQLite from Python by using the PySQLite module, available from http://pysqlite.sourceforge.net/. PySQLite provides an interface to SQLite compliant with the Python Database API Specification 2.0, so the programming interface should be familiar if you have used Python with another database in the past and should still be intuitive if not.

The Python Database API Specification can be found at http://www.python.org/peps/pep-0249.html.

Preparing to Use the Python Interface

Before you can install PySQLite, you need to be sure your system has the Python interpreter and the development libraries installed, as well as a C compiler. You also need to have a SQLite development librarylibsqlite.so or sqlite.dllinstalled, as PySQLite does not include it.

Some Linux distributions package the Python interpreter and development libraries separately, and simply selecting Python at install time may not have included the development package. On Red Hat Linux, for example, you need to make sure the python-devel package has been installed.

Download the PySQLite source from http://sourceforge.net/projects/pysqlite/ and extract it. The version number in the filename may be different from that shown.

$ gunzip -c pysqlite-0.5.1.tar.gz | tar xf -

Enter the pysqlite directory that has been created and issue the following command to compile PySQLite:

```
$ cd pysqlite
$ python setup.py build
running build
running build_py
creating build
creating build/lib.linux-i686-2.3
creating build/lib.linux-i686-2.3/sqlite
copying sqlite/__init__.py -> build/lib.linux-i686-2.3/sqlite
copying sqlite/main.py -> build/lib.linux-i686-2.3/sqlite
running build_ext
building '_sqlite' extension
[...]
```

The following step installs the PySQLite extension into system directories and so you will need to become the root user, if you are not already, using the su command.

```
# python setup.py install
running install
running build
running build_py
running build_ext
running install_lib
[...]
```

To test that PySQLite has installed correctly, run the test suite that is shipped with the source code. The following output indicates a successful installation:

```
# python tests/all_tests.py
..........................................................................
.....................................
----------------------------------------------------------------------
Ran 122 tests in 0.289s

OK
```

Using the Python Interface

Now that you have the PySQLite interface installed, let's look at how the interface is used from a Python script. In this section we'll see how to open and close databases and issue commands to SQLite.

Opening and Closing a Database

In order to use the PySQLite functionality, Python must first import the sqlite module. Then you can call the sqlite.connect() constructor method in order to open a database. The following is an example with Python running in interactive mode:

```
$ python
Python 2.3.3 (#1, May  7 2004, 10:31:40)
[GCC 3.3.3 20040412 (Red Hat Linux 3.3.3-7)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sqlite
>>> cx = sqlite.connect("pydb")
>>>
```

In its simplest form, the constructor takes a single database name parameter. The filename given can include a relative or absolute path and will be read from the current working directory if no path is specified. If that file exists, that database is opened; otherwise, a new empty database is created with the given filename. A connection object is returned, assigned to cx in this example.

If you followed the preceding example and no error occurred, exit Python and you'll see that the file pydb has been created in the current directory.

To close the connection to a database, simply call the .close() method on the connection object.

```
>>> cx.close()
```

After the connection has been closed, cx becomes unusable and an Error exception will be raised if you attempt to perform any operation on it.

Executing SQL Commands

To perform any database operation you must first create a cursor object using a valid database connection object. SQLite does not use the concept of cursors internally; however, PySQLite emulates this behavior in its interface in order to provide compliance with the Python Database API Specification. This is done using the .cursor() method, as follows:

```
>>> cx = sqlite.connect("pydb")
>>> cu = cx.cursor()
```

Note

You can actually create as many cursor objects as you like on the same database connection, although there is usually little point in doing so. Separate cursors do not isolate datachanges made in one cursor are immediately reflected in other cursors created from the same connection.

With a cursor object, you can use the .execute() method to execute an SQL command. Listing 9.1 shows a simple Python script that opens the pydb database and issues a CREATE TABLE statement to create the contacts table.

Part III: SQLite Administration

Chapter 10. General Database Administration

This chapter deals with SQLite administration topics. We will look at how the database parameters can be adjusted with the PRAGMA command and discuss ways to back up and restore your database. We will also examine some of the database internals that will help you to gain a better understanding of the SQLite engine.

The PRAGMA Command

The PRAGMA command provides an interface to modify the operation of the SQLite library and perform low-level operations to retrieve information about the connected database.

Fetching Database Information

This section describes the set of PRAGMA directives that allows you to find information about the currently attached databases.

```
PRAGMA database_list;
```

One row is returned for each open database containing the path to the database file and the name that the database was attached with. The first two rows returned will be the main database and the location in which temporary tables are stored.

A PRAGMA command can be executed from the sqlite program as well as through any of the language APIsafter all, the sqlite program is just a simple front end that passes typed commands to the C/C++ interface and uses a choice of callback function, selected using .mode, to output the rows to screen.

The following example shows the result of executing PRAGMA database_list through the sqlite program after a second database has been attached to the session:

```
sqlite> ATTACH DATABASE db2 AS db2;
sqlite> PRAGMA database_list;
seq   name   file
0     main   /home/chris/sqlite/admin/db1
1     temp   /var/tmp/sqlite_6x5rZ2drAEtVpzj
2     db2    /home/chris/sqlite/admin/db2
```

If executed from a language interface, the pseudo-table returned by this pragma contains columns named seq, name, and file.

To find information about a database table, use the table_info pragma with the following syntax:

```
PRAGMA table_info(table-name);
```

The table-name argument is required and can refer only to a table in the main database, not one attached using ATTACH DATABASE. One row is returned for each column in that table, containing the columns shown in Table 10.1.

Table 10.1. Columns in the Pseudo-Table Returned by PRAGMA table_info

cid	An integer column ID, beginning at zero, that shows the order in which columns appear in the table.
name	The name of the column. The capitalization used in the CREATE TABLE statement is retained.
type	The data type of the column, taken verbatim from the CREATE TABLE statement.
notnull	If the column was declared as NOT NULL, this column

Backing Up and Restoring Data

It is, of course, essential to back up your data regularly. SQLite stores databases to the filesystem, so backing up your databases can be as easy as taking a copy of the database files themselves and putting them somewhere safe.

However, if you are copying the database file, can you be sure that it is not being written to at the time you issue the copy command? If there's even a small chance that a SQLite write operation could happen during the copy, you should not use this method to back up your database to ensure that data corruption cannot occur.

SQLite implements locking at the database level using a lock on the database file itself, which works to the extent that other processes using the SQLite library know when the database is being written to. Many processes can read from a database at the same time; however, a single writer process locks the entire database, preventing any other read or write operation taking place.

On Windows platforms, if SQLite has locked the file the operating system will acknowledge the lock and will not allow you to copy the file until SQLite is done with it. Likewise the copy operation will lock the database file so that SQLite cannot access it. However, on Unix systems, file locks are advisoryin other words they tell you that the file is locked if you care to check, but do not otherwise prevent access. Therefore it is entirely possible to initiate a cp command while SQLite has a database file locked for writing.

The .dump Command

The .dump command in sqlite provides a simple way to create a backup file in the form of a list of SQL statements that can be used to re-create the database from scratch. The CREATE TABLE statements are preserved and one INSERT command is written for each row of data in the table.

Note

Because the .dump command is part of an application that uses the SQLite library, there are no issues with file locking to worry about. If the database is locked at the moment you attempt to fetch the rows from the database, the busy handler will be invoked, which, in sqlite, is to wait for the specified timeout before returning an error message.

A backup operation can therefore be performed as follows:

```
$ sqlite dbname .dump > backup.sql
```

Future restoration from the generated backup.sql is simple:

```
$ sqlite dbname < backup.sql
```

You can optionally pass a table name to .dump to extract only the schema and records for that table. The following example shows the dump file produced for the contacts table:

```
$ echo '.dump contacts'| sqlite db1
BEGIN TRANSACTION;
CREATE TABLE contacts (
  id INTEGER PRIMARY KEY,
  first_name CHAR,
  last_name  CHAR,
  email      CHAR);
INSERT INTO contacts VALUES(1,'Chris','Newman','chris@lightwood.net');
INSERT INTO contacts VALUES(2,'Paddy','O''Brien','paddy@irish.com');
INSERT INTO contacts VALUES(3,'Tom','Thomas','tom@tom.com');
INSERT INTO contacts VALUES(4,'Bill','Williams','bill@bill.com');
INSERT INTO contacts VALUES(5,'Jo','Jones','jo@jojones.com');
```

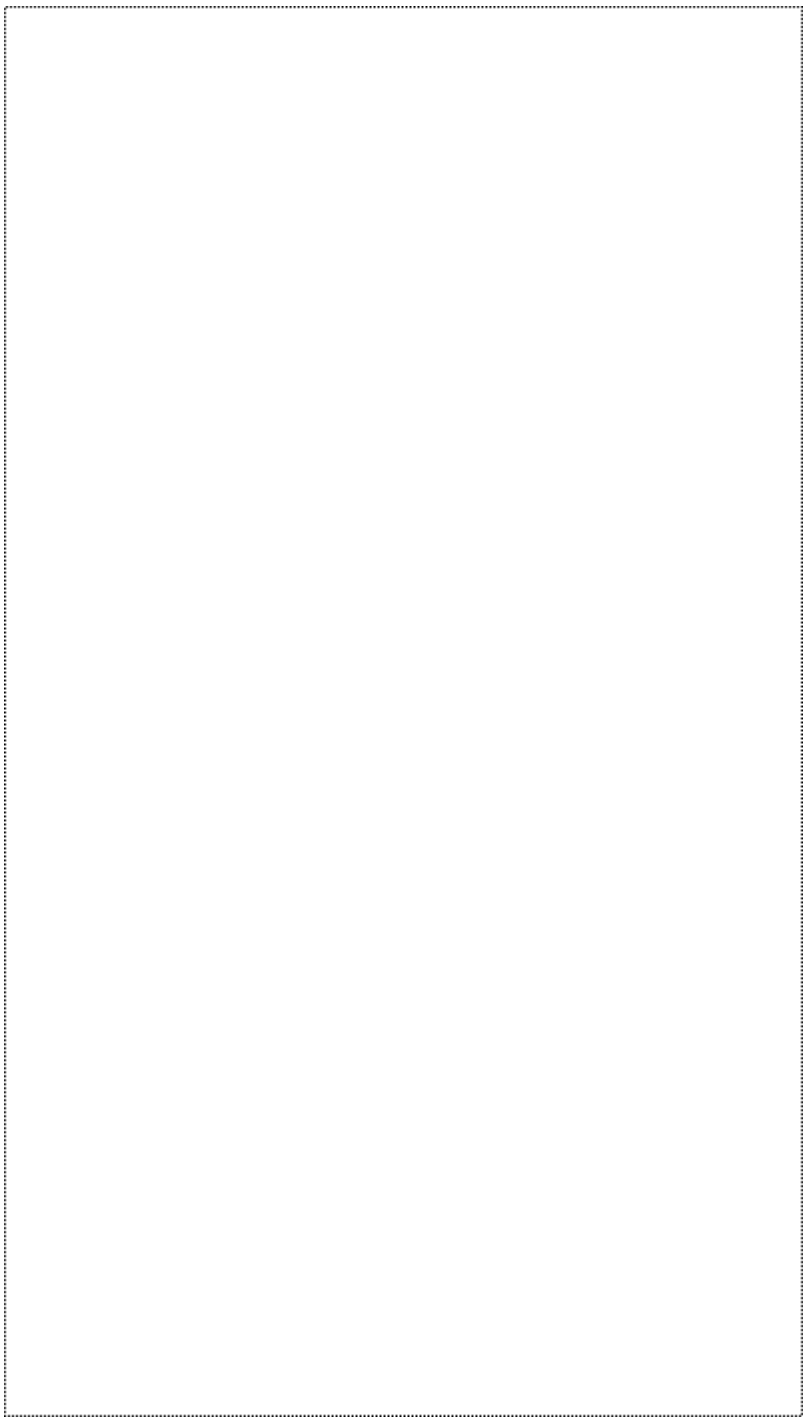
Exploring the SQLite Virtual Database Engine

In this section we'll take a look under the hood of SQLite to see how the command process works and to examine the internals of the Virtual Database Engine (VDBE). Though it is not essential to know what goes on behind the scenes, a little insight into how the SQLite engine processes your queries can help you understand SQLite better, and if you want to gain a deeper knowledge of how SQLite is implemented, you will learn the roles of the different source code files.

SQLite Architecture

Let's begin by looking at the steps SQLite goes through in order to process a query. Figure 10.1 shows a diagram of the flow of information between submitting a query for processing and the output being returned.

Figure 10.1. Architecture of the SQLite Database Engine



Interface

Access to the Database File

Because SQLite uses a local filesystem to store its databases, the SQL language does not implement the GRANT and REVOKE commands that are commonly found in client/server database systems. They simply do not make any sense in this context.

Therefore the only access issues that need to be addressed are those associated with the database file itself and are done at the operating-system level.

File Permissions

For SQLite to be able to open a database, the user under which a process is running must have permission to access that file on the operating system.

Quite simply, to open a database for reading the process must be able to open the file in read-only mode, and to open it for writing the process must have both read and write access to the file.

On a Unix system, if the process runs as the owner of the database file, the file mode can be 0400 or 0600 for read and write access respectively. Mode 0444 would enable all users to access the database file as read-only, whereas mode 0644 would additionally grant write access to the owner of the file, and mode 0666 would allow writing by all users respectively. More information about file permissions can be found from man chmod.

Locking and Timeouts

Remember that although there is no limit to the number of concurrent reads allowed by SQLite, a single write process will lock the database and prevent any further read or write operations until it is complete.

It is therefore important to handle the SQLITE_BUSY return code if more than one process might access your database at a time. The more instances of SQLite there are running in your application, the greater chance that one of them will encounter a timeout when trying to access the database file.

The simplest approach is to continue attempting the query in a loop until SQLITE_OK is returned or some other exit forces you to stop trying, with a small pause on each iteration of the loop. Listing 10.3 shows how you might take care of this.

Listing 10.3. Handling the SQLITE_BUSY Status
[View full width]

```
#include <stdio.h>
#include <sqlite.h>

int main()
{

  char *errmsg;
  int ret;

  sqlite *db = sqlite_open("db1", 0777, &errmsg);

  if (db == 0)
  {
    fprintf(stderr, "Could not open database: %s\n", errmsg);
    sqlite_freemem(errmsg);
    return(1);
  }

  do {
    ret = sqlite_exec(db, "INSERT INTO mytable (col1, col2) VALUES (1, 2)", NULL, NULL,
➥ &errmsg);
```

Part IV: Appendixes

Appendix A. Downloading and Installing SQLite

This appendix will show you where to go online to download the latest version of SQLite, how to pick the correct distribution, andif you decide on a source code installationhow to compile it.

Obtaining SQLite

The download page for SQLite is http://www.sqlite.org/download.html and is split into three sections:

- Precompiled Binaries for Linux
- Precompiled Binaries for Windows
- Source Code

Unless you have a particular requirement to build SQLite from source, a binary distribution will work fine for Linux and Windows systems.

Version numbers shown in this appendix are current at the time of writing; however, you should check for the latest stable version.

RPM Installation for Linux

The easiest way to install SQLite on compatible Linux systems is to use the RPM packages. The rpm command is found on the RedHat and Fedora Core Linux distributions, though other distributions may also support it.

There are two RPM packages available for SQLite, found in the precompiled binaries section of the download page.

sqlite-2.8.15-1.i386.rpm contains the shared library required to run dynamically linked SQLite applications, and the sqlite program.

sqlite-devel-2.8.15-1.i386.rpm contains the static library, header files, and documentation in the form of man pages.

You should obtain both packages and install using the following steps as the root user:

```
# rpm -ivh sqlite-2.8.15-1.i386.rpm
Preparing...                ########################################### [100%]
   1:sqlite                 ########################################### [100%]
# rpm -ivh sqlite-devel-2.8.15-1.i386.rpm
Preparing...                ########################################### [100%]
   1:sqlite-devel           ########################################### [100%]
```

The sqlite package must be installed before sqlite-devel to satisfy the package dependencies.

Binary Installation for Linux

The non-RPM binary distribution of SQLite is split into three files.

sqlite-2.8.15.bin.gz is a statically linked version of the sqlite program. This can be used alone to access and create SQLite databases from the command line.

sqlite-2.8.15.so.gz is the shared library that is needed to compile programs using the C/C++ interface.

tclsqlite-2.8.15.so.gz is a library file containing the TCL bindings for SQLite as well as the C/C++ interface.

Installation of each package is done using the same basic stepsuncompress the file, move it to a suitable location on your system, and set the appropriate file permission.

Appendix B. Command Reference for the sqlite Tool

This appendix lists the commands that can be used within the sqlite monitor program.

Dot Commands

The dot commands for sqlite can be used to change the output format, fetch information about the database, and modify some settings. In this appendix, each dot command is shown first followed by an explanation.

Obtaining a List of Dot Commands

`.help`

Displays the full list of dot commands for the installed sqlite with brief descriptions.

Changing the Output Format

`.mode list`

The default output mode. Displays one line per record, with each column separated by a specific character or string. The default separator is the pipe character (|).

`.separator` string

Changes the separator for list mode output to string.

`.mode lines`

Causes sqlite to output each column in the result of a query to be displayed on a line by itself, with the value prefixed by the name of the column. Subsequent records are separated by a blank line.

`.mode columns`

Displays one line per record with data aligned in fixed-width columns.

`.width` width1 width2 ... widthN

Specifies the width in characters for each column in turn, where width1 is the leftmost column returned by the query.

A width setting of 0 (the default) will automatically size the column to whichever is largest of: the length of the column heading, the length of the value in the first row of data, or 10 characters.

`.headers` on|off

Determines whether column headings are displayed in column output mode.

`.mode insert` table-name

Causes a list of full INSERT statements to be generated for the records returned by the query. The table-name argument determines the name of the table for the INSERT statements.

`.mode html`

The output is generated as an XHTML table with one set of <TR> tags and each column as a <TD> element. If

Appendix C. SQL Syntax Reference

This appendix summarizes the SQL language understood by SQLite. It provides the syntax for all supported SQL statements.

Naming Conventions

A standard identifier name must begin with a letter or an underscore character, and may contain any number of alphanumeric characters or underscores. No other characters can be used. There is no enforced upper limit on the length of an identifier name. Names can be as long as you like, but don't make them so long that you dread having to type them in full each time.

Square brackets or double quotes can be used to indicate a non-standard identifier name to the SQLite parser. Identifier names enclosed in this way can include characters other than the underscore, including spaces and even other square brackets, and this also allows you to use SQL keywords as identifiers.

Note

It is not generally a good idea to use non-standard identifier names in your database; although, square brackets can still be used around standard identifier names without consequence if you are familiar with this syntax from SQL Server or Microsoft Access.

Reserved Keywords

Tables C.1 through C.3 list the reserved keywords in SQLite. Table C.1 shows the fallback keywords, which can be used as identifiers without being delimited.

Table C.1. Fallback Keywords in SQLite

ABORT	AFTER	ASC	ATTACH	BEFORE
BEGIN	DEFERRED	CASCADE	CLUSTER	CONFLICT
COPY	CROSS	DATABASE	DELIMITERS	DESC
DETACH	EACH	END	EXPLAIN	FAIL
FOR	FULL	IGNORE	IMMEDIATE	INITIALLY
INNER	INSTEAD	KEY	LEFT	MATCH
NATURAL	OF	OFFSET	OUTER	PRAGMA
RAISE	REPLACE	RESTRICT	RIGHT	ROW
STATEMENT	TEMP	TEMPORARY	TRIGGER	VACUUM
VIEW				

Table C.3. System Object Names in SQLite

ROWID	MAIN	OID	ROWID	SQLITE_MASTER

SQL Command Syntax

This section details the SQL command syntax understood by SQLite. For clarity SQL keywords are shown in uppercase; however, SQLite is not case sensitive. Keywords and identifiers can be typed in uppercase, lowercase, or mixed case, and different capitalizations of the same string can be used interchangeably.

Creating and Dropping Database Objects

The CREATE object and DROP object statements are used to create and drop database objects.

CREATE TABLE

To create a new database table, use CREATE TABLE.

```
CREATE [TEMP | TEMPORARY] TABLE table-name (
    column-def [, column-def]*
    [, constraint]*
)
```

A column in the CREATE TABLE statement is defined as follows:

```
name [type] [[CONSTRAINT name] column-constraint]*
```

To drop a table, use DROP TABLE.

```
DROP TABLE [database-name.] table-name
```

Column Constraints

The optional column-constraint is composed of one or more of these keywords: NOT NULL, DEFAULT, UNIQUE, PRIMARY KEY, CHECK, and COLLATE.

```
NOT NULL [ conflict-clause ]
```

NOT NULL enforces that the column must always contain a value. An error will be raised on any attempt to insert a NULL value into the column.

```
DEFAULT value
```

DEFAULT defines a value that the column should take if no value is given when a row is inserted.

```
UNIQUE [ conflict-clause ]
```

UNIQUE creates a UNIQUE INDEX on the column, ensuring that the same value cannot be entered into this column more than once. There can be more than one UNIQUE INDEX on a table if required.

```
PRIMARY KEY [sort-order] [ conflict-clause ]
```

PRIMARY KEY creates a UNIQUE INDEX on the column designated as primary key for the table. Additionally if the column type is INTEGER, this column is used internally as the actual key of the table and the value is assigned automatically by SQLite if it is not specified when a row is inserted. Only one PRIMARY KEY can be specified on each table.

ANSI SQL Commands and Features Not Supported

The SQL language implemented by SQLite is fairly comprehensive; however, a few commands and features of the ANSI-92 specification are not available.

ALTER TABLE

SQLite does not allow the schema of a table to be changed, so the ALTER TABLE command is not supported.

To add, remove, or modify columns in a table, you must drop the table and re-create it with the revised schema. The usual way to do this is with a temporary table to hold the data from the old table while it is being re-created.

COUNT(DISTINCT column-name)

The DISTINCT keyword cannot appear inside a COUNT function. Instead you must use a nested subquery.

```
SELECT COUNT(DISTINCT mycol) FROM mytable;
```

becomes

```
SELECT COUNT(*) FROM (
  SELECT DISTINCT mycol FROM mytable
);
```

GRANT and REVOKE

Because access to SQLite databases takes place at the filesystem level, the only permissions that can be applied are those available to the underlying operating system. Therefore the GRANT and REVOKE commands are meaningless for SQLite.

INSERT, UPDATE, and DELETE on Views

SQLite does not allow write actions to be performed directly on a view, even if there is only one base table in the view. However, a trigger can be created on a view using the INSTEAD OF syntax, in which you can perform the appropriate INSERT, UPDATE, or DELETE on the underlying table(s).

RIGHT OUTER JOIN

LEFT OUTER JOIN is implemented, but not RIGHT OUTER JOIN. Therefore the table order in your queries must allow outer joins to be performed in this direction.

CHECK and FOREIGN KEY Constraints

Although the SQL syntax allows CHECK and FOREIGN KEY clauses to be included, they are ignored. They may be implemented in a future version.

Trigger Limitations

SQLite does not support the FOR EACH STATEMENT type of trigger, or INSTEAD OF TRiggers on tables. INSTEAD OF TRiggers can only be used on views.

Nested Transactions

Only one transaction can be active at a time. The name argument to BEGIN TRANSACTION is ignored.

Appendix D. PHP Interface Reference

This appendix lists the PHP functions that can be used to communicate with a SQLite database.

Further information and examples of usage submitted by users can often be found in the annotated PHP manual at http://www.php.net/manual/en/ref.sqlite.php.

Predefined Constants

Functions that return an array of results can take an optional result_type argument to determine what type of array is created. These are the valid constants:

- SQLITE_ASSOC causes the array to use the string type column name as the array index.
- SQLITE_NUM causes the array to use a numerical index starting from zero for each column in the result.
- SQLITE_BOTH causes the array to use both string and numerical keys.

If no constant is specified, SQLITE_BOTH is assumed.

Runtime Configuration

In php.ini the sqlite.assoc_case value affects the case of column names used for key values in associative arrays. It can take the following values:

- 0 Mixed case
- 1 Uppercase
- 2 Lowercase

The default behavior is to use mixed-case keys reflecting the natural case of the column headings. Using a setting of 1 or 2 will cause the case of the keys to be converted to uppercase or lowercase respectively.

Using this option incurs a slight performance penalty, but if case-folding of array keys is required, it is much faster to do so at this level than to implement it in your code.

Function Reference

In this section, SQLite functions and their results are described and organized by function category. The function is given first, followed by its description.

Opening and Closing a Database

```
resource sqlite_open ( string filename [, int mode [, string &error_message]])
```

Opens the database specified by filenameeither in the current directory or referenced via a relative or absolute pathand returns a database connection resource.

```
resource sqlite_popen ( string filename [, int mode [, string &error_message]])
```

Opens a persistent connection to the specified database file. To open an in-memory database, use :memory: as the filename.

Note

When the sqlite_popen() function is run in a web environment, file permissions must allow the web server user to gain read and write access to both the database file itself and its directory so that journal files can be written.

```
void sqlite_close ( resource dbhandle)
```

Closes a database connection.

```
void sqlite_busy_timeout ( resource dbhandle, int milliseconds)
```

Specifies the duration in milliseconds for which SQLite should wait for a lock to clear on the database file before failing with an SQLITE_BUSY return code. The default value is 60 seconds (60,000 milliseconds).

Executing a Query

```
resource sqlite_query ( resource dbhandle, string query)
resource sqlite_query ( string query, resource dbhandle)
```

Causes SQLite to execute the given query and returns a seekable result set resource.

Note

The arguments to sqlite_query() and other such functions can be specified in either order for consistency with both other SQLite interfaces and other PHP extensions. The preferred ordering is to specify the database resource firstthe order used by other SQLite extensions.

```
resource sqlite_unbuffered_query ( resource dbhandle, string query)
resource sqlite_unbuffered_query ( string query, resource dbhandle)
```

Works similarly to sqlite_query() but returns a result resource that can only be accessed sequentially. Unless random access is required, this function should be used as it provides much better performance.

```
string sqlite escape string ( string item)
```

Appendix E. C Interface Reference

This appendix lists the C library functions that can be used to communicate with a SQLite database.

The Core API

The core interface to the SQLite library is considered to be just three functions that allow you to open and close a database and to execute a query using a user-defined callback function. In this section we'll also look at the error codes returned from the core API.

Opening and Closing a Database

You can open and close a database as follows:

```
sqlite *sqlite_open(
  const char *dbname,
  int mode,
  char **errmsg
);

void sqlite_close(sqlite *db);
```

The return value of sqlite_open() and the argument to sqlite_close() is an opaque sqlite data structure.

```
typedef struct sqlite sqlite;
```

Executing a Query

You can execute a query as follows:

```
int sqlite_exec(
  sqlite *db,
  char *sql,
  int (*xCallback)(void *, int, char **, char **),
  void pArg,
  char **errmsg
);
```

The callback function has the following prototype:

```
int callback(void *pArg, int argc, char **argv, char **columnNames) {
  return 0;
}
```

Error Codes

This section describes SQLite error codes; each error code is followed by a description of its meaning.

```
#define SQLITE_OK           0   /* Successful result */
```

Returned if everything worked and there were no errors.

```
#define SQLITE_ERROR        1   /* SQL error or missing database */
```

Indicates an error in the SQL statement being executed.

```
#define SQLITE_INTERNAL     2   /* An internal logic error in SQLite */
```

The Non-Callback API

The non-callback API provides an alternative way to retrieve data from a SQLite database by compiling an SQL statement into a virtual machine of type sqlite_vm:

```
typedef struct sqlite_vm sqlite_vm;
```

Creating a Virtual Machine

You can create a SQLite virtual machine as follows:

```
int sqlite_compile(
  sqlite *db,               /* The open database */
  const char *zSql,         /* SQL statement to be compiled */
  const char **pzTail,      /* OUT: uncompiled tail of zSql */
  sqlite_vm **ppVm,         /* OUT: the virtual machine to execute zSql */
  char **pzErrmsg           /* OUT: Error message. */
);
```

The return code from sqlite_compile() is SQLITE_OK if the operation is successful; otherwise, one of the error codes listed in the preceding example is returned.

Step-by-Step Execution of an SQL Statement

Each invocation of sqlite_step() for a virtual machine, except the last one, returns a single row of the result:

```
int sqlite_step(
  sqlite_vm *pVm,           /* The virtual machine to execute */
  int *pN,                  /* OUT: Number of columns in result */
  const char ***pazValue,   /* OUT: Column data */
  const char ***pazColName /* OUT: Column names and datatypes */
);

int sqlite_finalize(
  sqlite_vm *pVm,           /* The virtual machine to be finalized */
  char **pzErrMsg           /* OUT: Error message */
);
```

Return Codes

The return code from sqlite_step() can be SQLITE_BUSY, SQLITE_ERROR, SQLITE_MISUSE, or either of the following.

```
#define SQLITE_ROW         100  /* sqlite_step() has another row ready */
```

Indicates that another row of result data is available.

```
#define SQLITE_DONE        101  /* sqlite_step() has finished executing */
```

Indicates that the SQL statement has been completely executed and sqlite_finalize() should now be called.

The return code from sqlite_finalize() indicates the overall success of the SQL command and will be the same as if the query had been executed using sqlite_exec().

The Extended API

The extended API provides a range of non-core functions to assist with development of software that uses an embedded SQLite database.

Finding Information About the SQLite Library

```
const char sqlite_version[];
```

Contains the current library version number.

```
const char sqlite_encoding[];
```

Contains the current library encoding version.

Finding Information About Changes to the Database

Several functions can return information about changes that have been made to the database.

```
int sqlite_last_insert_rowid(sqlite *db);
```

Returns the most recently assigned autoincrementing value of an INTEGER PRIMARY KEY field.

```
int sqlite_changes(sqlite *db);
```

Returns the number of rows affected by an UPDATE or DELETE statement.

Checking SQL Statements

```
int sqlite_complete(const char *sql);
```

Returns trUE if a complete SQL statement is provided, that is, the statement ends with a semicolon. Returns FALSE if more characters are required.

Interrupting an SQL Statement

```
void sqlite_interrupt(sqlite *db);
```

Causes the current database operation to exist at the first opportunity, returning SQLITE_INTERRUPT to the calling function.

Convenience Functions

The following function fetches the entire result of a database query with a single function call:

```
int sqlite_get_table(
  sqlite *db,
  char *sql,
  char ***result,
  int *nrow,
  int *ncolumn,
  char **errmsg
);
```

Adding New SQL Functions

SQLite allows you to add new functions to the SQL language that can subsequently be used in your queries.

Registering Functions

```
int sqlite_create_function(
  sqlite *db,
  const char *zName,
  int nArg,
  void (*xFunc)(sqlite_func*,int,const char**),
  void *pUserData
);
```

Creates a regular function in SQL from the function pointed to by xFunc.

```
int sqlite_create_aggregate(
  sqlite *db,
  const char *zName,
  int nArg,
  void (*xStep)(sqlite_func*,int,const char**),
  void (*xFinalize)(sqlite_func*),
  void *pUserData
);
```

Creates an aggregating function with function xStep executed once for each row returned by the query, and xFinalize invoked once after all rows have been returned.

The xFunc and xStep arguments are pointers to functions with the following prototype.

```
void xFunc(
  sqlite_func *context,
  int argc,
  const char **argv
);
```

The finalize function requires only the context argument.

```
void xFinalize(sqlite_func *context);
```

The context argument is an opaque data type sqlite_func.

```
typedef struct sqlite_func sqlite_func;
```

Setting Return Values

```
char *sqlite_set_result_string(
  sqlite_func *p,
  const char *zResult,
  int n
);

void sqlite_set_result_int(
  sqlite_func *p,
  int iResult
);

void sqlite_set_result_double(
```

Appendix F. Perl Interface Reference

This appendix is a reference for the Perl DBI module, which can be used to communicate with a SQLite database. Methods and attributes specific to the DBD::SQLite module are also listed.

In this appendix, the commands or methods are shown first, followed by the explanation.

The Perl DBI

The Perl interface to SQLite is the Perl Database Interface (DBI) module using the SQLite Database Driver (DBD) module.

```
use DBI;
```

Loads the DBI module into your script. There is no need to explicitly load a DBD driver as DBI takes care of this automatically.

In this appendix we use $dbh to refer to a database handle object and $sth for a statement handle.

Opening and Closing a Database

```
$dbh = DBI->connect($data_source, $username, $auth, \%attr);
```

Opens a database connection using the arguments given and creates a database handle object $dbh.

The data_source argument for a SQLite connection takes the following form, where dbfile can refer to a file in the current working directory or may contain a relative or absolute path.

```
DBI:SQLite2:dbname=dbfile
```

The username and auth arguments are not required for a SQLite database and can be omitted or passed as empty strings.

The optional attr argument can be used to set certain database handle attributes, described later in this appendix.

```
$rc = $dbh->disconnect();
```

Closes an open database connection, returning a Boolean $rc value.

Executing SQL Statements

The following methods can be called on a statement handle:

```
$sth = $dbh->prepare($sql);
$sth = $dbh->prepare($sql, \%attr);
```

Prepares a statement, $sql, for execution by the database and returns a statement handle object, $sth.

The optional attr argument can be used to set certain statement handle attributes, described later in this appendix.

```
$sth->execute();
$sth->execute(@bind_values);
```

Executes a prepared SQL statement. Return value $rv will be the number of rows affected, or -1 if not known. If zero rows are affected, the return value is 0E0, which is treated as a zero value but regarded as true. A zero return code indicates statement failure.

```
$dbh->do($sql);
$dbh->do($sql, @bind_values);
```

Appendix G. Tcl Interface Reference

This appendix lists the Tcl commands that are used to communicate with a SQLite database.

The Tcl Library

The Tcl library for SQLite is called libtclsqlite.so on Unix platforms and sqlite.dll on Windows systems. The library file should reside in a location that your scripts will be able to load from. Typically this would be a subdirectory of /usr/share/tcl on Unix or C:\tcl\lib on Windows.

In this appendix, commands from the Tcl library are shown first, followed by the explanation.

```
package require sqlite
```

Imports the sqlite package into a Tcl script.

Opening and Closing a Database

`sqlite dbcmd` database-name

Opens a database called database-nameeither from the current directory or referenced via a relative or absolute pathcreating it if it does not already exist. On success, a new command called dbcmd is registered in Tcl, upon which the methods described in the rest of this appendix can be applied.

```
dbcmd close
```

Closes the database connection and destroys dbcmd.

Executing a Query

```
dbcmd eval query
```

Causes SQLite to execute the given query and returns the entire data set fetched as a single list.

`dbcmd eval query array {` code-block `}`

As SQLite executes the query, for each row fetched an element in array is created for every column returned by the query and the commands in code-block are executed.

Additionally the data types of the returned columns are stored as elements named typeof:column-name in array, and the list of columns returned can be found in array(*).

`dbcmd eval query {} {` code-block `}`

If the empty string is used in place of array, code-block is still executed once for each row in the dataset; however, the fetched columns are stored to scalar variables with the same name as their respective columns.

Convenience Functions

`dbcmd onecolumn` query

Causes SQLite to execute the given query as eval; however, it returns only the first column from the first row of the dataset.

Finding Information About a Query

Appendix H. Python Interface Reference

This appendix is a reference for the PySQLite extension, which provides an interface to SQLite that is compliant with the Python Database Specification 2.0.

Before attempting any database connectivity with PySQLite, you must import the sqlite module, using

```
import sqlite
```

In this appendix, the commands are shown first, followed by the explanation.

Opening and Closing a Database

```
cx = sqlite.connect(database, mode=0755, converters={}, autocommit=0,
                    encoding=None, timeout=None, command_logfile=None)
```

Opens a SQLite connection to database and returns a connection object to cx.

The mode argument is currently ignored but may be used in the future to specify the file mode with which the database file is opened.

The converters argument can be used to specify mappings from SQL data types using Python conversion functions.

Set autocommit to 1 if you want PySQLite to commit each SQL command immediately when executed, rather than the default behavior of batching groups of INSERT, UPDATE, and DELETE statements into a single transaction.

The encoding argument allows you to specify the encoding to be used on Unicode strings. Its value can be noneif only ASCII characters are to be acceptedor utf-8.

The default timeout value of None means that if the database file is locked, SQLite will return an error immediately. Setting a value in seconds instructs the database engine to keep trying to obtain a lock for the given amount of time.

Use command_logfile with a file object argument to specify a file to which all SQL statements executed through PySQLite will be written.

```
cx.close()
```

Closes a database connection and rolls back any uncommitted transactions. The connection object is unusable after .close() has been called.

Executing SQL Statements

All SQL statements must be executed through a cursor object.

```
cu = cx.cursor()
```

Creates a new cursor object for connection cx.

```
cu.execute(sql)
cu.execute(sql, args)
```

Passes sql to SQLite for execution. Formatted strings are permitted in sql using the standard Python format codes. Unsafe characters do not require delimiting if passed as format arguments.

```
cx.commit()
```

Issues a COMMIT TRANSACTION command to store any unsaved changes to the database. A .commit() operation is always required to save changes when autocommit is off.

```
cx.rollback()
```

Issues a ROLLBACK TRANSACTION command to reject any unsaved changes.

```
numrows = cu.rowcount
```

Creating User-Defined Functions

```
cx.create_function(name, argc, func)
```

Registers a user-defined function called name in SQL using the Python function func. The number of arguments to the function is given in argc.

```
cx.create_aggregate(name, argc, aggregate)
```

Registers a user-defined aggregating function called name in SQL using the Python class aggregate, which must contain the member functions reset(), step(), and finalize().

Error Handling

PySQLite extends the StandardError class with its own Error class, with which errors related to SQLite operations can be handled.

Two subclasses of Error are implementedInterfaceError and DatabaseError. These allow you to detect whether the error originates with the SQLite engine or the PySQLite interface.

The DatabaseError class is further subdivided as follows:

- DataError
- OperationalError
- IntegrityError
- InternalError
- ProgrammingError
- NotSupportedError

Appendix I. The Future of SQLite

This appendix looks at how SQLite might be improved, enhanced, or extended in the future.

SQLite Version 3.0

At the time of writing, SQLite version 3.0 was close to release, and it may very well be available as a stable release by the time you are reading this book.

The decision was made to cover the latest 2.8 version throughout the book primarily because of the vast user base this version of SQLite already has.

Although version 3.0 adds some exciting functionality, it also has a completely different API, so existing users will be slow to migrate, if they switch to the new version at all. New users should consider carefully whether they want to use the new version or stick with a tried and tested library for their application.

New features will no longer be added to SQLite 2.8, but it will continue to be supported and have maintenance fixes issued for the foreseeable future, so do not be put off from using it simply because a new version has been released.

Let's take a look at some of the changes in SQLite version 3.0.

Naming Changes

Because it is important that SQLite 2.8 can continue to be used while the new version is introduced, version 3.0 uses a new naming convention and allows both versions of the SQLite library to be linked to the same program if required.

The library itself has been renamed to libsqlite3.so on Unix and sqlite3.dll on Windows, and the include file is now called sqlite3.h. The sqlite command-line tool has also been renamed to sqlite3.

Within the API itself, function names have also been changed to be prefixed with sqlite3, but the names themselves remain familiar; for example, sqlite3_open() and sqlite3_changes() will behave as expected.

Although most of the sqlite3 prefixed function names have the same prototype as their corresponding functions in SQLite 2, one significant difference is the sqlite3_open() function, which works as follows:

```
int sqlite3_open(const char *filename, sqlite3 **ppDb)
```

In SQLite 3, the return value from the database connection function is SQLITE_OK on success or the corresponding error code on failure. The database handle is returned in *ppDb.

The full list of version 3 API calls can be found at http://www.sqlite.org/capi3ref.html.

File Format Changes

The database file format has been overhauled. Version 2.8 databases cannot be read by the 3.0 library, and the reverse is true.

The new file format uses a B+Tree data structure for table storage to replace the B-Trees, which allows better scalability within a filesystem-based database. It is also more highly optimized through omitting unused fields from the disk image and better encoding of floating-point numbers, which has been shown to produce a 2535% reduction in the overall file size.

Migrating from the old database file format to SQLite 3.0 is very simple if you have both the sqlite and sqlite3 command-line tools installed. The following command would effectively upgrade olddb to the new file format, saving the new version as newdb.

```
$ sqlite olddb .dump | sqlite3 newdb
```

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index