CHAPTER 2

Getting Started

On the whole, it's easy to begin using SQLite no matter what operating system you are using. For the vast majority of users, SQLite can be installed and running with a new database in hand in under 5 minutes, regardless of experience. This chapter covers everything you need to know in order to install SQLite and work with databases. You will have a working knowledge of where to obtain SQLite software or source code and how to install or compile it on multiple platforms. By the time you finish this chapter, you will have a new SQLite database with tables, views, and indexes that you can query, back up, and restore. Furthermore, you will learn everything you need to know about managing SQLite databases, including how to create, view, and examine their contents. Finally, you will be introduced to several tools with which to work with SQLite in various environments. This chapter does include some examples that use SQL to introduce the SQLite command-line program. If you are not yet familiar with SQL, you should still be able to follow the examples without much trouble. SQL is addressed in detail in Chapter 4.

Where to Get SQLite

The SQLite website (`www.sqlite.org`) provides both precompiled binaries of SQLite as well as source code. Binaries are available for both Windows and Linux.

There are several binary packages to choose from, each of which is specific to a particular way of using SQLite. The binary packages are as follows:

- **Statically linked command-line program (CLP).** This version of the SQLite command-line program has the database engine compiled in and is a self-contained, standalone program. This provides a convenient way to work with SQLite databases from the command line without having to worry about whether or not the SQLite shared library is installed on your system or located in the right place.

- **SQLite dynamic link library (DLL).** This is the SQLite database engine packaged into a shared library, or Windows DLL. Use this with programs that dynamically link to SQLite. This form makes it easier to upgrade SQLite without having to recompile the software that depends on it.

- **Tcl extension.** This is a Tcl extension library that enables you to connect to SQLite from within the Tcl language. SQLite's author, Richard Hipp, happens to be the author and maintainer for the Tcl extension.

SQLite's source code is provided in two forms that vary by platform. One form is for compiling on Windows and the other is for compiling on POSIX platforms such as Linux, BSD, and Solaris. The source code itself does not differ between source distributions. Rather, the two distributions simply include different conveniences that make it easier to work within their respective environments. The Windows distribution, for example, has the preprocessing and code generation performed by GNU Autoconf and other associated Unix build tools already added to the source files, freeing Windows users from having to bother with them.

SQLite on Windows

Whether you are using SQLite as an end user, or you are writing programs that use it, SQLite can be installed on Windows with a minimum of fuss. In this section, we will cover all the options—from installing the available binary packages to building everything from source using the most popular compilers. Let's start with the easy things first and progress to things more technically challenging.

Getting the Command-Line Program

The SQLite command-line program (hereafter referred to as the CLP) is by far the easiest way to get started using SQLite. Follow these steps to obtain the CLP:

1. Open your favorite browser and navigate to the SQLite home page: `www.sqlite.org`.
2. Click the download link on the top right of the page. This will take you to the download page.
3. Under the section *Precompiled Binaries For Windows*, there should be a file whose name is of the form `sqlite-3_x_y.zip`, where *x* and *y* are the minor version numbers. There should be a comment beside it that reads *A command-line program for accessing and modifying SQLite databases*. Download this file to a temporary folder.
4. Unzip the file. In Windows XP and Me, you can right-click on the file and a context menu should appear. Select Extract All from the menu. This will bring up the Windows Extraction Wizard. Follow the instructions and extract the file to a folder of your choosing. The wizard will then place a copy of the statically linked CLP in the specified folder. The file's name should be `sqlite3.exe`. If you have an older version of Windows, you may need to obtain a compression utility, such as WinZip (`www.winzip.com`), to unzip the file. To run the CLP from any directory in the Windows shell, you need to copy it to a folder that is in your Windows system path. A suitable default that should work on all versions of Windows is the `\windows\system32` folder on your root partition (`C:\` for most systems).

■ Note If you don't know what your Windows system path is, here is how to find it. Click Start ➤ Control Panel. (If you are using Windows XP, look at the Control Panel dialog box at the left side of the window. It will either say "Switch to Classic View" or "Switch to Category View." If you see the former, then click it. This will put the view into Classic View.) Double-click the System icon. In the resulting dialog box, select the Advanced tab and click the Environmental Variables button. In the System Variables list box, double-click the Path entry. This will open the Edit System Variables dialog box. The Values text box contains a long list of paths delimited by semicolons. All of the folders listed here are part of your system path. You can add an additional folder to this path if you like by simply appending a semicolon to the end of the line and typing the new path.

5. Open a command shell. You can do this different ways, depending on your version of Windows. Using the Windows Start menu, select Start ➤ Run. Type `cmd` in the Open drop-down box (or `command` if you are using a version of Windows 98/Me). Click OK (Figure 2-1). This will open a Windows command shell. If this doesn't work, try going to Start ➤ All Programs ➤ Accessories ➤ Command Prompt.

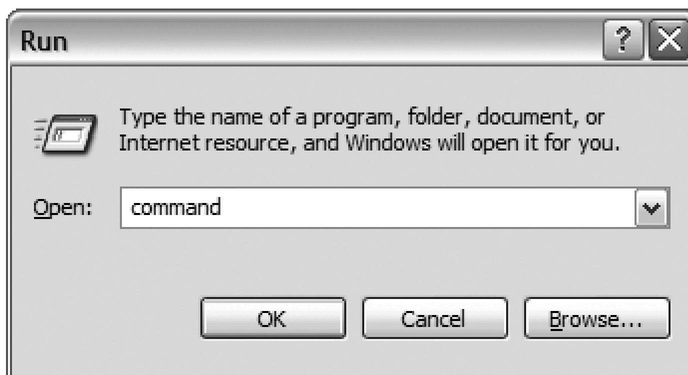

Figure 2-1. *Opening a Windows command shell*

6. Within the shell, type `sqlite3` on the command line and press Enter. This should bring up a SQLite command prompt. (If you get an error, then the `sqlite3.exe` executable has not been copied to a folder in your system path. Recheck your path, and place a copy of the program somewhere within it.) When the SQLite shell appears, type `.help` on the command line. This will display a list of commands with their associated descriptions similar to the one in Figure 2-2. Type `.exit` to exit the program. You now have a working copy of the SQLite CLP installed on your system.

```
C:\WINDOWS\system32\cmd.exe - sqlite3

C:\>sqlite3
SQLite version 3.3.4
Enter ".help" for instructions
sqlite> .help
.databases             List names and files of attached databases
.dump ?TABLE? ...      Dump the database in an SQL 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
.import FILE TABLE     Import data from FILE into TABLE
.indices TABLE         Show names of all indices on TABLE
.mode MODE ?TABLE?     Set output mode where MODE is one of:
                         csv      Comma-separated values
                         column   Left-aligned columns.  (See .width)
                         html     HTML <table> code
                         insert   SQL insert statements for TABLE
                         line     One value per line
                         list     Values delimited by .separator string
                         tabs     Tab-separated values
                         tcl      TCL list elements
.nullvalue STRING      Print STRING in place of NULL values
.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 used by output mode and .import
.show                  Show the current values for various settings
.tables ?PATTERN?      List names of tables matching a LIKE pattern
.timeout MS            Try opening locked tables for MS milliseconds
.width NUM NUM ...     Set column widths for "column" mode
sqlite> _
```

Figure 2-2. *The SQLite shell on Windows*

If you are especially eager to work with SQLite at this point, you may want to skip ahead to the section "The CLP in Shell Mode;" the next few sections are geared to developers who want to write programs that use SQLite.

Getting the SQLite DLL

The SQLite DLL is used for software that has been compiled to link dynamically to SQLite. Most software that uses SQLite in this fashion usually includes its own copy of the SQLite DLL and installs it automatically with the software.

If you are going to be programming with SQLite, using the DLL is probably the easiest way to start. The SQLite DLL can be obtained as follows:

1. Go to the SQLite home page, `www.sqlite.com`. On the upper right of the page, click the download link. This will take you to the download page.
2. On the download page, find the section *Precompiled Binaries For Windows*.
3. Locate the DLL zip file. This file will have the description *This is a DLL of the SQLite library without the TCL bindings*. The filename will have the form `sqlitedll-3_x_y.zip`, where *x* and *y* are the minor versions. If you want Tcl support included, select the file with the name of the form `tclsqlitedll-3_x_y.zip`.

4. Download and unzip the file. The extracted contents should include two files: the actual DLL file (`sqlite3.dll`) accompanied by another file called `sqlite3.def`. The SQLite DLL provided here is thread safe. That is, it was compiled with the `THREADSAFE` preprocessor flag defined. You can therefore use this DLL in multithreaded programs.

In order to use the DLL, it needs to be either in the same folder with programs that use it or placed somewhere in the system's path (see the note on the Windows System path in the previous section).

If you want to write programs that use the SQLite DLL, you will need to create an import library with which to link your programs. This is quite simple to do using the `sqlite3.def` mentioned earlier. If you are using Microsoft Visual C++, open a shell, change the directory to the SQLite distribution, and simply run the command

```
LIB /DEF:sqlite3.def
```

If you are using MinGW (see the section "Building SQLite with MinGW" later in this chapter), run the command

```
dlltool --def sqlite3.def --dllname sqlite3.dll --output-lib sqlite3.lib
```

Running either of these commands will create an import library called `sqlite3.lib` with which you can link your programs. By linking your programs to this import library, they will load and use the SQLite DLL upon execution.

Compiling the SQLite Source Code on Windows

Building SQLite from source within Windows is straightforward. Depending on the compiler you are using and what you are trying to achieve, there are several approaches to compiling SQLite. The most common scenarios on Windows include using either Microsoft Visual C++ or MinGW. Both are addressed here. Information on how to compile SQLite with other compilers can be found on the SQLite Wiki (`www.sqlite.org/cvstrac/wiki?p=HowToCompile`).

The Stable Source Distribution

Stable versions of SQLite's source code can be obtained in zip files from the SQLite website. Bleeding-edge versions can be obtained from anonymous CVS. Unless you are familiar with CVS, using the source distribution is the easiest way to go. To download a stable source distribution, follow these steps:

1. Go to the SQLite website, `www.sqlite.org`. Follow the download link, which will take you to the download page.

2. On the download page, find the Source Code section.

3. The first two files should be zip files containing the source code for Windows. The file you want to download should have a name with the form `sqlite-source-3-x_y.zip`, where *x* and *y* are the minor version numbers. The important thing here is that you want `sqlite-source-3-x_y.zip`, which corresponds to SQLite version 3, not `sqlite-source2-x_y.zip`, which corresponds to SQLite version 2.

■ **Note** The Windows zip archive and the other (POSIX) tarballs on the download page differ slightly in their contents. While they contain identical source code, the SQLite distribution uses some POSIX build tools (`sed`, `awk`, etc.) to dynamically generate some C source code in the build process. These build tools are not available by default on Windows systems. Therefore, the Windows source archive includes all of the preprocessing and generated code as a matter of convenience to Windows users who lack the build support infrastructure of Unix. This is why Windows users should use the zip archives rather than the POSIX tarballs on the download page. It is still possible to build the tarballs on Windows, but you need the requisite POSIX build tools (which are included in the MSYS/MinGW distributions covered in a moment).

---

4. Extract/unzip the file to a directory of your choosing. The extracted contents will be the complete SQLite version 3 source code for Windows.

Anonymous CVS

If you want to play with the latest features or participate in SQLite development, then retrieving SQLite from anonymous CVS makes the most sense. Anonymous CVS provides read access to the CVS repository—you can check out the code but you cannot commit any changes you may make in your local copy back to the repository. CVS allows you to maintain the absolutely latest version of the SQLite source code. If you want, you can keep your copy of the code synced up to the day, hour, or minute to stay current with changes as they are committed. Thus, if you see an important bug fix or feature posted that you want to take advantage of, all you need to do is perform a CVS update and recompile your copy of the code. Also note that the version in CVS corresponds to that found in the POSIX tarballs: it still requires the code-generation and preprocessing steps before the code can be built under Windows. You must therefore have the requisite POSIX build tools mentioned in the previous note in order to build SQLite from CVS on Windows.

Obtaining SQLite from CVS on Windows is perhaps easiest by using WinCVS. WinCVS (shown in Figure 2-3) is a well-written graphical application that makes working with CVS repositories easy and intuitive. WinCVS can be obtained at `www.wincvs.org`.

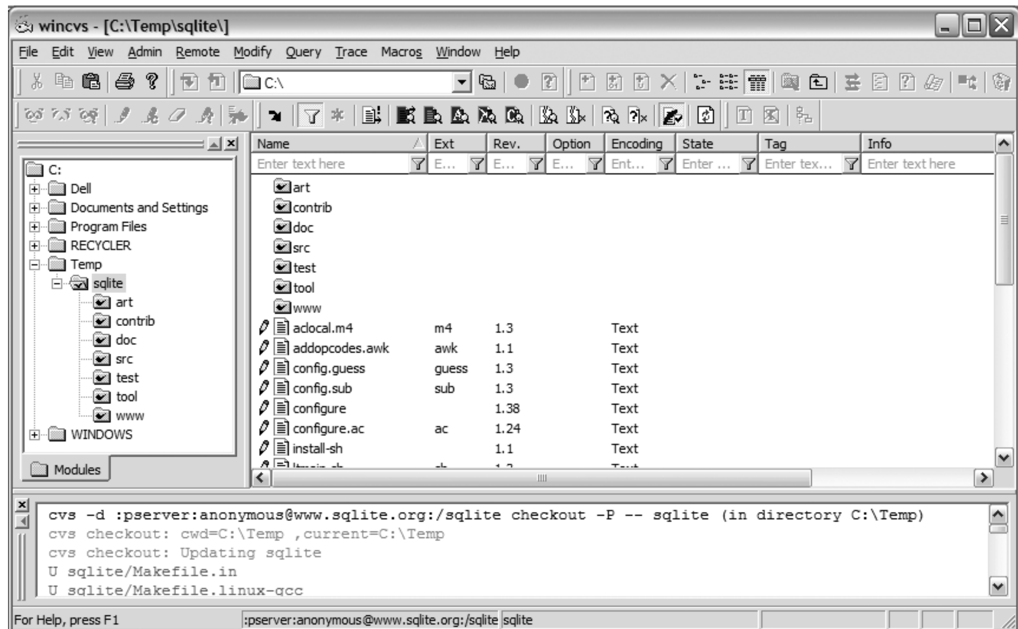

Figure 2-3. *WinCVS*

Once it's installed, you can configure WinCVS to connect to SQLite's CVS repository as follows:

1. Create a folder in which to check out the SQLite source code (e.g., `C:\Temp`).

2. Open WinCVS. On the left pane, navigate to the folder you created.

3. From the main menu, select Admin ➤ Login. In the CVSROOT text box, type `:pserver:anonymous@www.sqlite.org:/sqlite`, and click OK. This will bring up a dialog box requesting the home folder. Use the folder you created earlier (`C:\Temp` in this example). Next a password dialog box will appear. For the password, enter "anonymous", and click OK. If you log in successfully, then the output pane will display the following:

   ```
   cvs -d :pserver:anonymous@www.sqlite.org:/sqlite login
   Logging in to :pserver:anonymous@www.sqlite.org:2401:/sqlite

   ***** CVS exited normally with code 0 *****
   ```

4. In the main menu, select Remote ➤ Checkout Module (Figure 2-4). In the Module Name And Path On Server box, type `sqlite`. In the CVSROOT box, type `:pserver:anonymous@www.sqlite.org:/sqlite`.

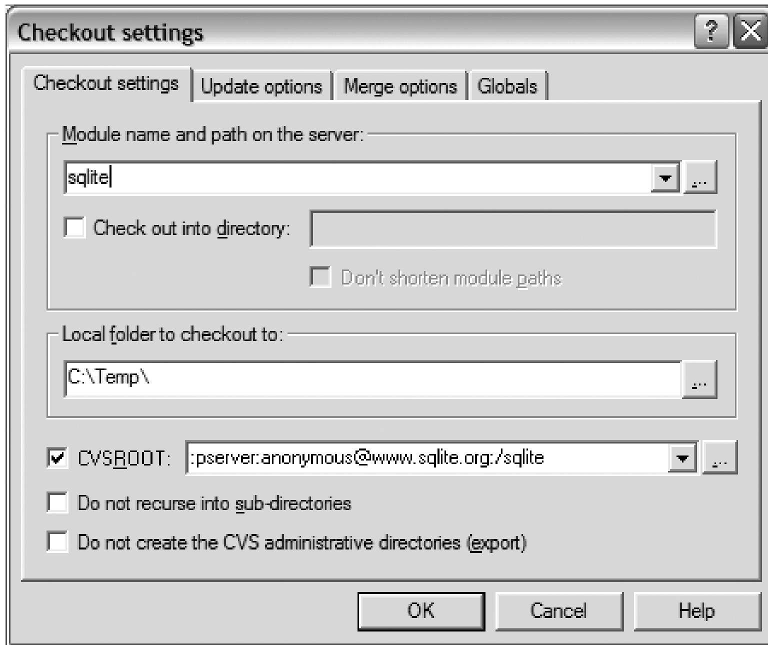

Figure 2-4. *WinCVS Checkout Settings dialog box*

5. Note that the Local Folder To Checkout To text box should already contain the path to the folder you created to store the source code. Click OK. You should see a long list of files appear in the output pane, the first part of which looks like this:

```
cvs -d :pserver:anonymous@www.sqlite.org:/sqlite checkout -P sqlite (in
directory C:\Temp\sqlite)
cvs checkout: cwd=C:\Temp\sqlite ,current=C:\Temp\sqlite
cvs checkout: Updating sqlite
U sqlite/Makefile.in
U sqlite/Makefile.linux-gcc
U sqlite/README
U sqlite/VERSION
U sqlite/aclocal.m4
```

Once completed, the latest version of SQLite should be checked out in your local SQLite folder.

Building the SQLite DLL with Microsoft Visual C++

To build the SQLite DLL from source using Visual C++, follow these steps:

1. Start Visual Studio. Create a new DLL project within the unpacked SQLite source directory. Do this by going to File ➤ New ➤ Project. Under Project Types (Figure 2-5), select Visual C++ Projects, and then select Win32. Choose the Win32 Project template. In the Location text box, enter the folder name that contains your SQLite source folder. In this example, it would be `C:\Temp`. In the Name text box, enter the name of the folder containing the SQLite source code—`sqlite` in this example. This will create the Visual C++ project inside the existing SQLite source folder (`C:\Temp\sqlite`). Click OK.

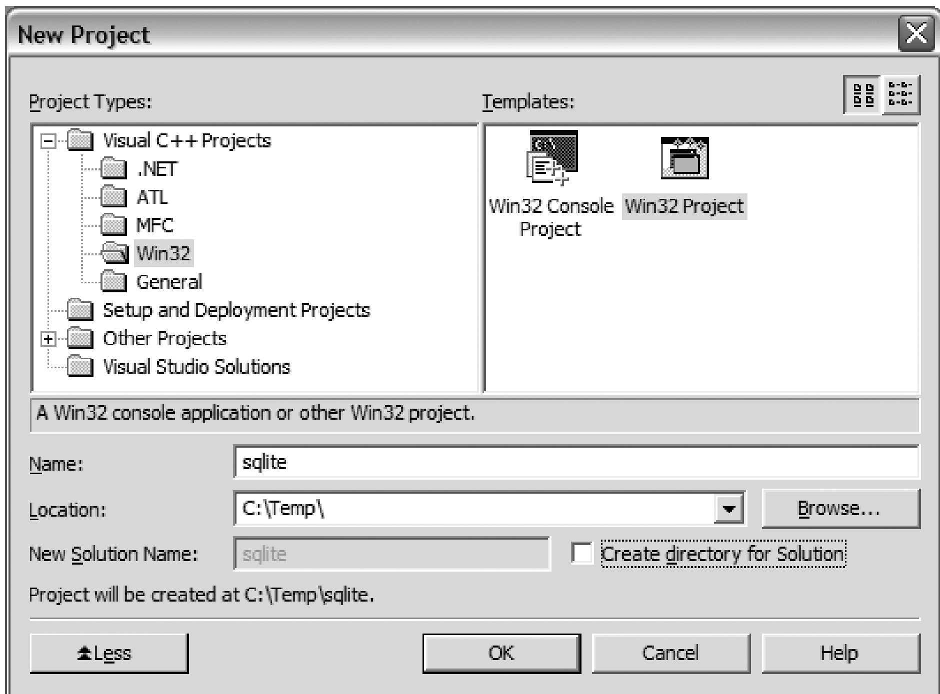

Figure 2-5. *Creating a new Visual C++ project*

2. Next, the Win 32 Application Wizard will automatically open (Figure 2-6). Choose Application Settings and set the application type to DLL. Be sure to check the Empty Project box. Click Finish, and this will create a blank DLL project.

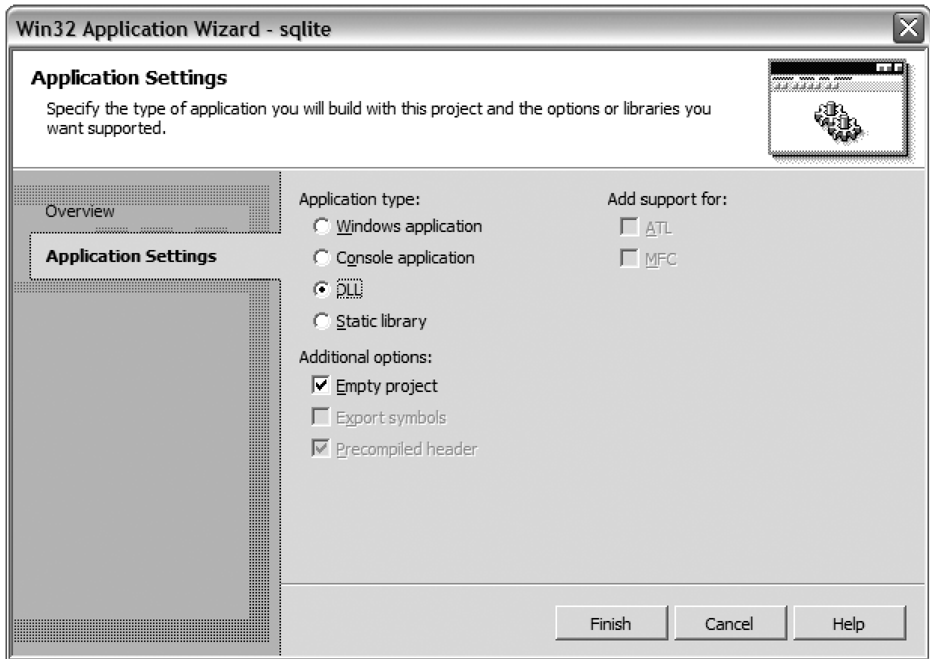

Figure 2-6. *The Win32 Application Wizard*

3. Add SQLite source files and headers to the project. Select Project ➤ Add Existing Item. Add all `.c` and `.h` files in the directory except for two files: `tclsqlite.c` and `shell.c`. (The first is for Tcl support; the second is for creating the SQLite shell, neither of which we want in this case.)

4. If you want to build the DLL with thread safety, you need to make sure that you have the preprocessor flag `THREADSAFE` defined in the project. To do this, select Project ➤ Properties and under the C/C++ item in the left tree view, select Preprocessor (Figure 2-7). Click on the Preprocessor Definitions cell. A small button will appear to the right. Click this button to open the dialog box shown in Figure 2-8. Add `THREADSAFE` to the bottom of the list and click OK. Also, you will need to make sure that you use the multithreaded Microsoft C runtime library DLL. Specify this by selecting the Code Generation item (Figure 2-7) and under Runtime Library select Multi-threaded Debug (`/MTd`).

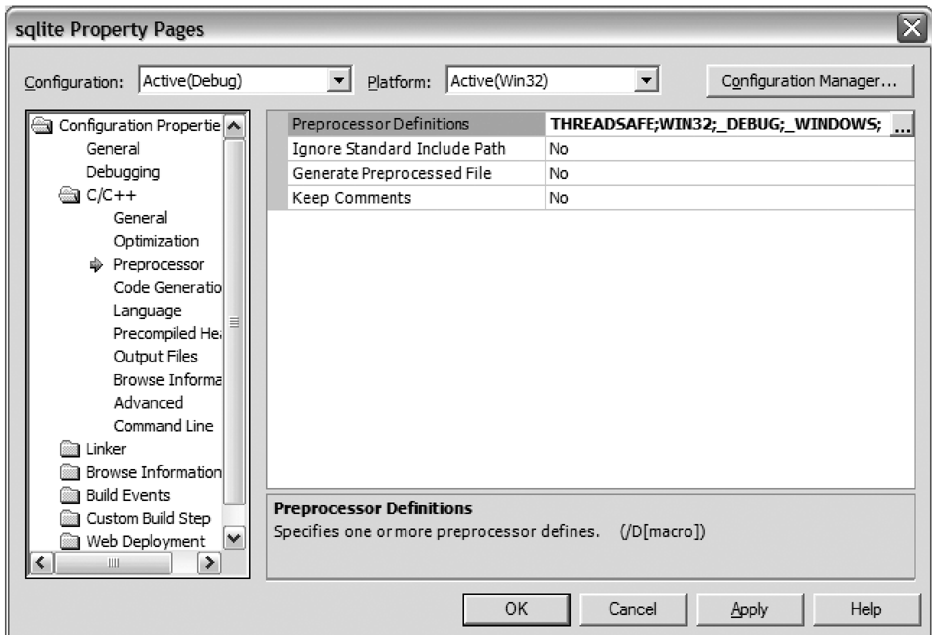

Figure 2-7. *Preprocessor project settings*

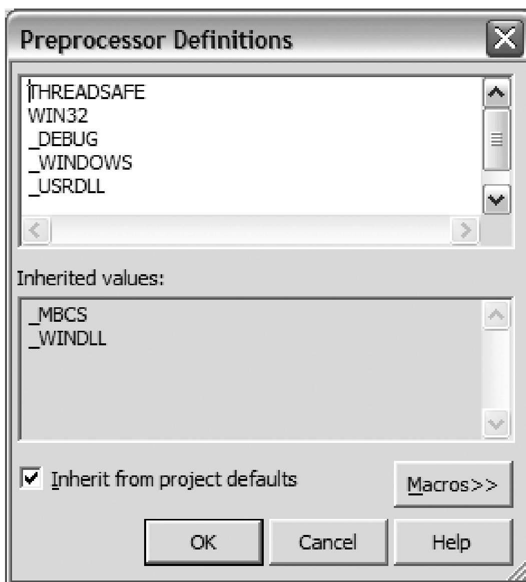

Figure 2-8. *Preprocessor definitions*

5. Specify an export or a module definition (`.def`) file. This file defines what symbols (or functions) to export (make visible) to programs that link to the library. SQLite's source distribution is kind enough to include such a file (`sqlite3.def`) for this very purpose. Also within the Property Pages dialog box, select All Configurations in the Configuration drop-down box (Figure 2-9). Then click the Linker folder and click the Input submenu. In the Module Definition File property page, type `sqlite3.def`. You are now ready to build the DLL.

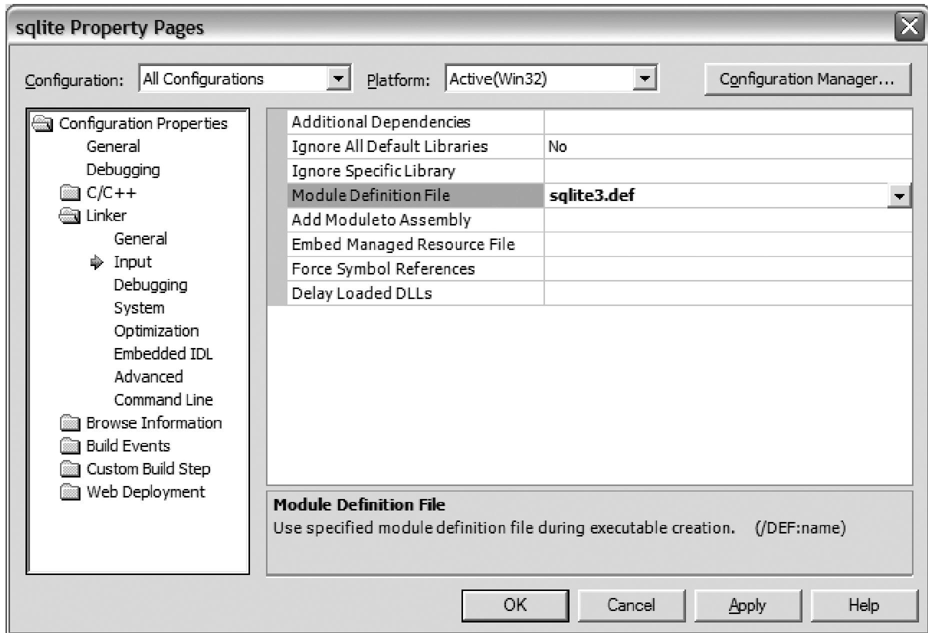

Figure 2-9. *Project properties*

6. From the main menu, select Build ➤ Build sqlite to build the DLL.

7. Once you have built the DLL, be sure to create the import library as described in the section "Getting the SQLite DLL."

Building a Dynamically Linked SQLite Client with Visual C++

The binary for a static CLP is available on the SQLite website, but what if you want a version that uses the SQLite DLL? To build such a version in Visual C++, do the following:

■ **Note** Many of the steps are very similar to the process of building a DLL, mentioned earlier—you may want to use some of the figures listed there for reference.

---

1. From the main menu, select File ➤ New ➤ Project. Under Project Types, select Visual C++ Projects, and then select Win32. Choose the Win32 Project template. Name the project (`shell`, for example) and click OK.
2. After this, the Win 32 Application Wizard will automatically open. Choose Application Settings and set the application type to Console Application, and be sure to check the Empty Project box. Click Finish to create a blank executable project.
3. Next you want to add the SQLite shell source file. Select Project ➤ Add Existing Item. In the dialog box that appears, add the source file `shell.c`.
4. Tell Visual C++ to link against the SQLite DLL. Select Project ➤ Properties. In the dialog box that appears, select All Configurations in the Configuration drop-down box. Next select the Linker folder. Select the Input submenu and within the Additional Dependencies property page, add `sqlite3.lib`. You are now ready to build the program. Note that the SQLite DLL needs to be either in the same directory as the command-line program or in the Windows system path.

■ **Note** If you build the SQLite DLL with threading enabled or you obtain the DLL from the SQLite website, you need to use the multithreaded Microsoft C runtime library DLL when building the CLP. To do this, refer to the second half of step 4 in "Building the SQLite DLL with Microsoft Visual C++." It contains two informative figures that make it easy to set this option.

Building SQLite with MinGW

MinGW (`www.mingw.org`) is a very nice distribution of the GNU Compiler Collection (GCC) for Windows. It also includes freely available Windows-specific header files and libraries that you can use to create native Windows programs that do not rely on any third-party C runtime DLLs. Put simply, it is a free C/C++ compiler for Windows, and a very good one at that. It is usually used in conjunction with MSYS, which is a portable POSIX environment that makes Unix users feel at home on Windows. Together, the two provide a powerful environment with which to compile and build software on Windows. With this build environment, you can build both the tarball archives as well as the source directly from CVS.

To build the SQLite DLL from source with MinGW, do the following:

1. Open your favorite browser and navigate to the MinGW website: `www.mingw.org`.
2. On the left side of the page, click the Download link, which will take you to the main download page.
3. On the download page, look for the FileList section. There will be a link to SF File Releases page, which at the time of this writing is located above the long listing of files. Follow that link. It will take you to the SourceForge download page for MinGW. Scroll down to the MinGW section and download the latest release (`MinGW-5.0.2.exe` at the time of this writing).

4. After downloading the file, click Back a few times to return to the initial SourceForge download page. Look for the MSYS section and download the current version (`MSYS-1.0.11-2004.04.30.exe` at the time of this writing).

5. Install MinGW, followed by MSYS. The MinGW installer will prompt you for a mirror site. Choose the one that is closest to you. In the next dialog box (the Choose Components dialog box shown in Figure 2-10), make sure you also select the second option—`g++ compiler`—in addition to the default (if you forget you can always rerun the MinGW installer and select this component). Select the defaults for all other screens. Double-click the respective files, which will invoke installers for each package. Follow the directions provided by the installers.

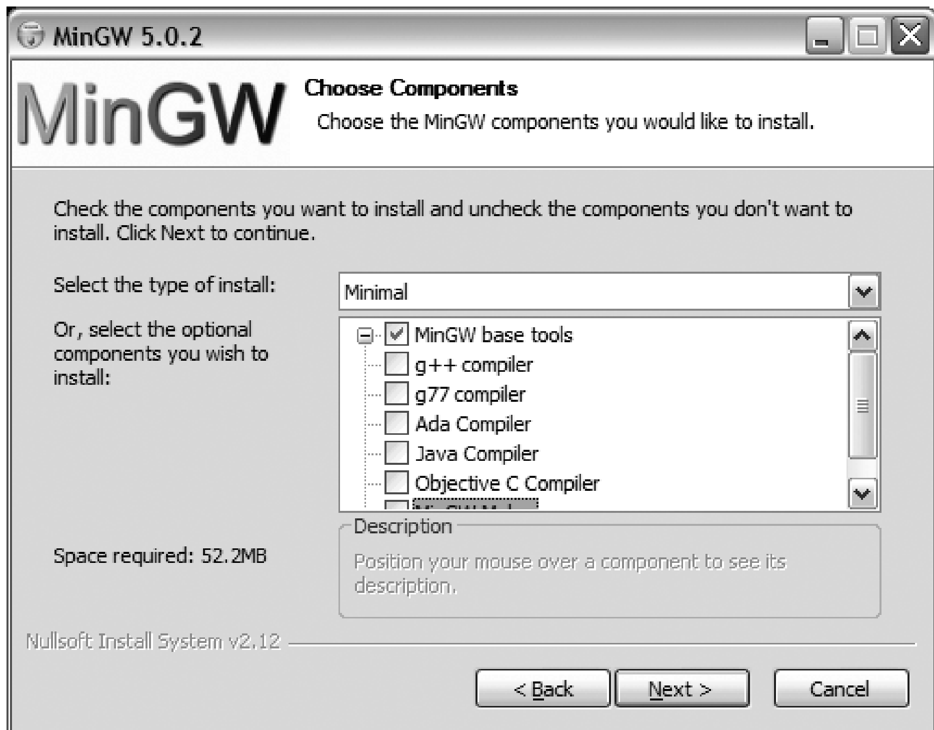

Figure 2-10. *MinGW installation components*

6. Download the Linux/Unix SQLite source code distribution. Navigate to `www.sqlite.org` and click on the download link. On the download page, find the Source Code section. The file you are looking for is the POSIX source distribution and should have a name of the form `sqlite-3.x.y.tar.gz`, where *x* and *y* are the minor version numbers (at the time of this writing, the current filename is `sqlite-3.3.4.tar.gz`). In Unix parlance, this kind of source archive is called a *tarball* (akin to a Windows zip file). Download the tarball and place it in a temporary directory (e.g., `C:\Temp`).

7. MSYS will have placed an icon on your desktop. Double-click that icon to open the environment.

8. Navigate to your temporary directory in which you downloaded the SQLite source distribution. Since this is a Unix-like environment, you will need to use Unix file system conventions. For example, to get to `c:\Temp`, you would type `cd /c/Temp`.

9. Unpack the SQLite tarball. Issue this command:

```
tar -xzvf sqlite-3.3.4.tar.gz
```

10. Move into the unpacked directory:

```
cd sqlite-3.3.4
```

11. Create the Makefile. For a single-threaded DLL, run

```
./configure
```

12. If you want to create a multithreaded DLL, run

```
./configure --enable-threads
```

13. Build the source:

```
make
```

14. Create the SQLite DLL:

```
dllwrap --dllname sqlite3.dll --def sqlite3.def *.o
```

15. Create the import library:

```
dlltool --def sqlite3.def --dllname sqlite3.dll --output-lib sqlite3.lib
```

You now have a functional SQLite DLL and import library. To build a version of the SQLite CLP that links against the SQLite DLL you just created, within the MSYS environment run the command

```
gcc -I . src/shell.c -o sqlite3.exe sqlite3.lib
```

Now from Windows Explorer, navigate to the temporary folder and double-click on `sqlite3.exe`. You now have a working SQLite CLP, which uses the SQLite DLL you created.

SQLite on POSIX Systems

SQLite compiles and builds identically on POSIX systems such as Linux, Mac OS X, FreeBSD, NetBSD, OpenBSD, Solaris, and others. SQLite binaries can be obtained in a variety of ways depending on the particular operating system.

Binaries and Packages

If you are using Mac OS 10.4 ("Tiger") or greater, you already have SQLite installed on your system. If not, there are several routes you can take to install it. The easiest way is to use one of the following Mac-specific package management systems, all of which include packages or ports for SQLite:

- **Metadistribution.** Metadistribution is a Gentoo-based package management system based on Gentoo's Portage. All packages are fetched from the Internet and built from source using a single command, `emerge`. You can find the Mac version at `www.metadistribution/macos/`.
- **Fink.** Fink is a Debian-based package management system that uses Debian utilities such as `dpkg`, `dselect`, and `apt-get`, in addition to its own utility—`fink`. You can download Fink from `http://fink.sourceforge.net`. With Fink, it is possible to install straight from precompiled binaries. No compilation step is needed.
- **Darwin Ports.** Darwin Ports is a package system written in Tcl, which like Metadistribution, installs from source. More information can be obtained from `http://darwinports.opendarwin.org`.

BSD users will have no trouble installing SQLite either. FreeBSD, OpenBSD, and NetBSD all have packages and/or ports for SQLite, all of which are very easy to install. As I write this, each distribution has ports for very recent versions of SQLite 3.*x*.

Solaris 10 uses SQLite as part of the OS; however at the time of this writing it uses version SQLite 2.*x*. An easy way to install SQLite in Solaris (and other open source software as well) is to use Blastwave's `pkg-get` utility. Blastwave is a Debian-based package management system for Solaris that makes installing free software on Solaris simple. You can get more information on Blastwave from `www.blastwave.org`.

As mentioned earlier, binaries for Linux are available directly from the SQLite website. The download page on SQLite's website provides the following binaries:

- **Statically linked command-line program.** The filename is of the form `sqlite3-3.`*`x`*`.`*`y`*`.bin.gz`, where *x* and *y* are the minor version numbers.
- **Shared library.** Two forms of the shared library exist. One form includes the Tcl bindings; the other does not. The name of the shared library with Tcl bindings has the form `tclsqlite-3.3.`*`x`*`.`*`y`*`.so.gz`, where again *x* and *y* are the minor version numbers. The name of the shared library without Tcl bindings is of the form `sqlite-3.`*`x`*`.`*`y`*`.so.gz`. Note that the shared libraries provided are not thread safe. If you need a thread-safe version, you will have to compile the library from source. See the section "Compiling SQLite from Source" for more details.
- **SQLite Analyzer.** This is a command-line program that provides detailed information about the contents of a SQLite database. You'll find information on this program in the section "Getting Database File Information."

Various RPM-based Linux distributions may include RPMs for SQLite, but may not include the most recent versions. Fedora and Mandriva don't appear to include RPMs for SQLite at all. Perhaps the best way to find the most recent versions of SQLite in RPM form is to consult the Internet. RPM PBone Search (`http://rpm.pbone.net`) and RPMFind (`http://rpmfind.net`) are two good places to start.

Debian-based distributions will have no trouble getting up-to-date versions of SQLite. SQLite 3 packages are available online in both Ubuntu and Debian repositories, among others.

Compiling SQLite from Source

Compiling SQLite from source on POSIX systems follows very closely the MinGW instructions given earlier for the Windows platform (actually it is more the other way around; MinGW installation apes Linux source installation!). To build SQLite on POSIX systems, you need to ensure that you have the GNU Compiler Collection (GCC) installed, including Autoconf, Automake, and Libtool. Most of the systems already discussed include all of these by default. With this software in place, you can build SQLite by doing the following:

1. Download the Linux/Unix SQLite tarball (source code) from the SQLite website. At the time of this writing, the current version is `sqlite-3.3.4.tar.gz`. Place it in a directory (e.g., `/tmp`).
2. Navigate to your build directory:
   ```
   cd /tmp
   ```
3. Unpack the SQLite tarball:
   ```
   tar -xzvf sqlite-3.3.4.tar.gz
   ```
4. Move into the unpacked directory:
   ```
   cd sqlite-3.3.4
   ```
5. Create the Makefile:
   ```
   ./configure
   ```
6. If you want to create a multithreaded shared library, run
   ```
   ./configure --enable-threads
   ```
7. Other options, such as the installation directory, are also available. For a complete list of configure options, run
   ```
   ./configure --help
   ```
8. Build the source:
   ```
   make
   ```
9. As root, install:
   ```
   make install
   ```

You now have a functional SQLite installation on your system that includes both the SQLite shared library and a dynamically linked CLP (which uses the SQLite shared library). If you have GNU Readline installed on you system, the CLP should be compiled with Readline support. Test it out by running it from the command line:

```
root@linux # sqlite3
```

This will invoke the CLP using an in-memory database. Type `.help` for a list of shell commands. Type `.exit` to close the application, or press Ctrl+D.

Working with SQLite Databases

The SQLite CLP is the most common means you can use to work with and manage SQLite databases. It runs on as many platforms as the SQLite library, so learning how to use it ensures you will always have a common and familiar way to manage your databases. The CLP is really two programs in one. It can run from the command line to perform various administration tasks, or it can be run in shell mode and act as an interactive query processor.

The CLP in Shell Mode

Open a shell and change directory to some temporary folder—say `C:\Temp` if you are on Windows or `/tmp` if you're in Unix. This will be your current working directory. All files you create in the course of working with the shell will be created in this directory.

■ **Note** To get a command line on Windows, go to Start ➤ Programs ➤ Accessories ➤ Command Prompt.

To invoke the CLP as in shell mode, type `sqlite3` from a command line, followed by an optional database name. If you do not specify a database name, SQLite will use an in-memory database (the contents of which will be lost when the CLP exits).

Using the CLP as an interactive shell, you can issue queries, obtain schema information, import and export data, and perform other miscellaneous database tasks. The shell will consider any statement issued as a query, except for commands that begin with a period (`.`). These commands are reserved for specific shell operations, a complete list of which can be obtained by typing `.help` as shown:

```
mike@linux tmp $ sqlite3
SQLite version 3.3.4
Enter ".help" for instructions
sqlite> .h
```

```
.databases             List names and files of attached databases
.dump ?TABLE? ...      Dump the database in a SQL 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
.import FILE TABLE     Import data from FILE into TABLE
.indices TABLE         Show names of all indices on TABLE
.mode MODE ?TABLE?     Set output mode where MODE is one of:
                         csv      Comma-separated values
                         column   Left-aligned columns.  (See .width)
                         html     HTML <table> code
                         insert   SQL insert statements for TABLE
```

```
                         line     One value per line
                         list     Values delimited by .separator string
                         tabs     Tab-separated values
                         tcl      TCL list elements
.nullvalue STRING      Print STRING in place of NULL values
.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 used by output mode and .import
.show                  Show the current values for various settings
.tables ?PATTERN?      List names of tables matching a LIKE pattern
.timeout MS            Try opening locked tables for MS milliseconds
.width NUM NUM ...     Set column widths for "column" mode
```

```
sqlite>.exit
```

You can just as easily type `.h` for short. Many of the commands can be similarly abbreviated, such as `.e`—short for `.exit`—to exit the shell.

Let's start by creating a database that we will call `test.db`. From the command line, open the CLP in shell mode by typing the following:

```
sqlite3 test.db
```

Even though we have provided a database name, SQLite does not actually create the database (yet) if it doesn't already exist. SQLite will defer creating the database until you actually create something inside it, such as a table or view. The reason for this is so that you have the opportunity to set various permanent database settings (such as page size) before the database structure is committed to disk. Some settings such as page size and character encoding (UTF-8, UTF-16, etc.) cannot be changed once the database is created, so this interim is where you have a chance to specify them. We will go with the default settings here, so to actually create the database on disk, we need only to create a table. Issue the following statement from the shell:

```
sqlite> create table test (id integer primary key, value text);
```

Now you have a database file on disk called `test.db`, which contains one table called `test`. This table, as you can see, has two columns:

- A primary key column called `id`, which has an autoincrement attribute. Wherever you define a column of type integer primary key, SQLite will apply an autoincrement function for the column. That is, if no value is provided for the column in an `INSERT` statement, SQLite will automatically generate one by finding the next integer value specific to that column.

- A simple text field called `value`.

Let's add a few rows to the table:

```
sqlite> insert into test (value) values('eenie');
sqlite> insert into test (value) values('meenie');
sqlite> insert into test (value) values('miny');
sqlite> insert into test (value) values('mo');
```

Now fetch them back:

```
sqlite> .mode col
sqlite> .headers on
sqlite> SELECT * FROM test;
```

```
id          value
----------  ----------
1           eenie
2           meenie
3           miny
4           mo
```

The two commands preceding the `SELECT` statement (`.headers` and `.mode`) are used to improve the formatting a little (both of which are covered later). We can see that SQLite provided sequential integer values for the `id` column, which we did not provide in the `INSERT` statements. While on the topic of autoincrement columns, you might be interested to know that the value of the last inserted autoincrement value can be obtained using the SQL function `last_insert_rowid()`:

```
sqlite> select last_insert_rowid();
```

```
last_insert_rowid()
-------------------
4
```

Before we quit, let's add an index and a view to the database. These will come in handy in the illustrations that follow:

```
sqlite> create index test_idx on test (value);
sqlite> create view schema as select * from sqlite_master;
```

To exit the shell, issue the `.exit` command:

```
sqlite> .exit
C:\Temp>
```

On Windows, you can also terminate the shell by using the key sequence Ctrl+C. On Unix, you can use Ctrl+D.

Getting Database Schema Information

There are several shell commands for obtaining information about the contents of a database. You can retrieve a list of tables (and views) using `.tables [pattern]`, where `[pattern]` can be any pattern that the SQL `LIKE` operator understands (we cover `LIKE` in Chapter 4 if you are unfamiliar with it). All tables and views matching the given pattern will be returned. If no pattern is supplied, all tables and views are returned:

```
sqlite> .tables
```

```
schema test
```

Here we see our table named `test` and our view named `schema`. Similarly, indexes for a given table can be printed using `.indices [table name]`:

```
sqlite> .indices test
```

```
test_idx
```

Here we see the index we created earlier on `test`, called `test_idx`. The SQL definition or data definition language (DDL) for a table or view can be obtained using `.schema [table name]`. If no table name is provided, the SQL definitions of all database objects (tables, indexes, views, and indexes) are returned:

```
sqlite> .schema test
```

```
CREATE TABLE test (id integer primary key, value text);
CREATE INDEX test_idx on test (value);
```

```
sqlite> .schema
```

```
CREATE TABLE test (id integer primary key, value text);
CREATE VIEW schema as select * from sqlite_master;
CREATE INDEX test_idx on test (value);
```

More detailed schema information can be had from SQLite's one and only system view, `sqlite_master`. This view is a simple system catalog of sorts. Its schema is described in Table 2-1. Querying `sqlite_master` for our current database returns the following (don't forget to use the `.mode col` and `.headers on` commands first to manually set the column format and headers):

Table 2-1. *SQLite Master Table Schema*

Name	Description
type	The object's type (table, index, view, trigger)
name	The object's name
tbl_name	The table the object is associated with
rootpage	The object's root page index in the database (where it begins)
sql	The object's SQL definition (DDL)

```
sqlite> .mode col
sqlite> .headers on
sqlite> select type, name, tbl_name, sql from sqlite_master order by type;
```

```
type        name        tbl_name    sql
----------  ----------  ----------  -------------------------------------
index       test_idx    test        CREATE INDEX test_idx on test (value)
table       test        test        CREATE TABLE test (id integer primary
view        schema      schema      CREATE VIEW schema as select * from s
```

We see a complete inventory of `test.db` contents: one table, one index, and one view, each with their respective SQL definitions.

There are few additional commands for obtaining schema information through SQLite's `PRAGMA` commands, `table_info`, `index_info`, and `index_list`, which are covered in Chapter 4.

■ **Tip** Don't forget that most shells keep a history of the commands that you execute. To rerun a previous command, you can hit the Up Arrow key to scroll through your previous commands.

Exporting Data

You can export database objects to SQL format using the `.dump` command. Without any arguments, `.dump` will export the entire database. If you provide arguments, the shell interprets them as table names or views. Any tables or views matching the given arguments will be exported. Those that don't are simply ignored. In shell mode, the output from the `.dump` command is directed to the screen by default. If you want to redirect output to a file, use the `.output [filename]` command. This command redirects all output to the file filename. To restore output back to the screen, simply issue `.output stdout`. So, to export the current database to a file `file.sql`, you simply do the following:

```
sqlite> .output file.sql
sqlite> .dump
sqlite> .output stdout
```

This will create the file `file.sql` in your current working directory if it does not already exist. If a file by that name does exist, it will be overwritten.

By combining redirection with SQL and the various shell formatting options (covered later), you have a great deal of control over exporting data. You can export specific subsets of tables and views in various formats using the delimiter of your choice, which can later be imported using the `.import` command described next.

Importing Data

There are two ways to import data, depending on the format of the file to import. If the file is composed of SQL, you can use the `.read` command to import (execute) the file. If the file contains comma-separated values (CSV) or other delimited data, you can use the `.import [file][table]` command. This command will parse the specified file and attempt to insert it into the specified table. It does this by parsing each line in the file using the pipe character (`|`) as the delimiter and inserting the parsed columns into the table. Naturally, the number of parsed fields in the file should match up with the number of columns in the table. You can specify a different delimiter using the `.separator` command. To see the current value set for the separator, use the `.show` command. This will show all user-defined settings for the shell, among them the current default separator:

```
sqlite> .show
```

```
     echo: off
  explain: off
  headers: on
     mode: column
nullvalue: ""
   output: stdout
separator: "|"
    width:
```

The `.read` command is the way to import files created by the `.dump` command. Using `file.sql` created earlier as a backup, we can drop the existing database objects (the `test` table and `schema` view) and reimport it as follows:

```
sqlite> drop table test;
sqlite> drop view schema;
sqlite> .read file.csv
```

Formatting

The shell offers a number of formatting options. The simplest are `.echo`, which echoes the last run command after issuing a command, and `.headers`, which includes column names for queries when set to `on`. The text representation of null values can be set with `.nullvalue`. For instance, if you want null values to appear as `NULL`, simply issue the command `.nullvalue NULL`. By default, this value is an empty string. Also, the shell prompt can be changed using `.prompt [value]`:

```
sqlite> .prompt 'sqlite3> '
sqlite3>
```

Result data can be formatted several ways using the .mode command. The current options are csv, column, html, insert, line, list, tabs, and tcl, each of which is helpful in different ways. The default is .list. For instance, list mode displays results with the columns separated by the default separator. Thus, if you wanted to dump a table in a CSV format, you could do the following:

```
sqlite3> .output file.csv
sqlite3> .separator ,
sqlite3> select * from test;
sqlite3> .output stdout
```

The contents of file.csv are now

```
1,eenie
2,meenie
3,miny
4,mo
```

Actually, since there is a CSV mode already defined in the shell, it is just as easy to use it in this particular example instead:

```
sqlite3> .output file.csv
sqlite3> .mode csv
sqlite3> select * from test;
sqlite3> .output stdout
```

and obtain similar results. The difference is that CSV mode will wrap field values with double quotes, whereas list mode (the default) does not.

Putting It All Together

Combining the previous three sections on exporting, importing, and formatting data, we now have an easy way to export and import data in delimited form. For example, to export only the rows of the test table whose value fields start with the letter "m" to a file called test.csv in comma-separated values, do the following:

```
sqlite> .output text.csv
sqlite> .separator ,
sqlite> select * from test where value like 'm%';
sqlite> .output stdout
```

If you want to then import this CSV data into a similar table with the same structure as the test table (call it test2), do the following:

```
sqlite> create table test2(id integer primary key, value text);
sqlite> .import text.csv test2
```

The CLP, therefore, makes it easy to both import and export text-delimited data to and from the database.

The CLP in Command-Line Mode

The CLP can be used from the command line for tasks such as importing and exporting data, returning result sets, and performing general batch processing. It is ideal for use in shell scripts for automated database administration. To see what the CLP offers in command-line mode, invoke it from the shell (Windows or Unix) with the -help switch, as shown here:

```
mike@linux tmp $ sqlite3 -help
```

```
Usage: sqlite3 [OPTIONS] FILENAME [SQL]
Options are:
   -init filename       read/process named file
   -echo                print commands before execution
   -[no]header          turn headers on or off
   -column              set output mode to 'column'
   -html                set output mode to HTML
   -line                set output mode to 'line'
   -list                set output mode to 'list'
   -separator 'x'       set output field separator (|)
   -nullvalue 'text'    set text string for NULL values
   -version             show SQLite version
   -help                show this text, also show dot-commands
```

The CLP in command-line mode takes the following arguments:

- A list of options (optional)
- A database filename (required)
- A SQL command to execute (optional)

Most of the options control output formatting except for the init switch, which specifies a batch file of SQL commands to process. The database filename is required. The SQL command is optional with a few caveats.

There are actually two ways to invoke the CLP in command-line mode. The first is to provide a SQL command. However, "SQL command" is somewhat misleading as you can provide SQLite shell commands as well, such as .dump and .schema. Any valid SQL or SQLite shell command will do. When it's provided, SQLite will simply execute the specified command, print the result to standard output, and exit. For example, to dump the test.db database from the command line, issue the command

```
sqlite3 test.db .dump
```

To make it useful, we should redirect the output to a file:

```
sqlite3 test.db .dump > test.sql
```

The file test.sql now contains the complete human-readable (SQL) representation of test.db. Similarly, to select all records for the test table, issue

```
sqlite3 test.db "select * from test"
```

The second way to invoke the CLP in command-line mode is to redirect a file as an input stream. For instance, to create a new database `test2.db` from our database dump `test.sql`, do the following:

```
sqlite3 test2.db < test.sql
```

The CLP will read the file as standard input, then process and apply all SQL commands within it to the `test2.db` database file.

So, in order for command-line mode to be invoked, either a SQL command or an input stream must be provided to the CLP. To further illustrate this, yet another way to create a database from the `test.sql` file is to use the `init` option and provide the `test.sql` as an argument:

```
sqlite3 -init test.sql test3.db
```

The CLP will process `test.sql`, create the `test3.db` database, and then go into shell mode. Why? The invocation included no SQL command or input stream. To get around this, you need to provide a SQL command. For example:

```
sqlite3 -init test.sql test3.db .exit
```

The `.exit` command prompts the CLP to run in command-line mode and does as little as possible. All things considered, redirection is perhaps the easiest method for processing files from the command line.

Database Administration

All database administration tasks can be performed within the shell and on the command line. Typically, the command line is easier to use for many general administration tasks, but it is really a matter of taste. Many of the common database administration tasks have been touched upon through the examples provided thus far. For the sake of completeness, I will nevertheless list the most common tasks along with the typical ways of performing them.

Creating, Backing Up, and Dropping Databases

Backing up a database can be done in two ways, depending on the type of backup you desire. A SQL dump is perhaps the most portable form for keeping backups. The standard way to generate one is using the CLP `.dump` command. From the command line, this is done as follows:

```
sqlite3 test.db .dump > test.sql
```

Within the shell, you can redirect output to an external file, issue the command, and restore output to the screen as follows:

```
sqlite> .output file.sql
sqlite> .dump
sqlite> .exit
```

Likewise, importing a database is most easily done by providing the SQL dump as an input stream to the CLP:

```
sqlite3 test.db < test.sql
```

This assumes that test.db does not already exist. If it does, then things may still work if the contents of test.sql are different from those of test.db. You will of course get errors if test.sql contains objects that already reside within test.db.

Making a binary backup of a database is little more than a file copy. One small operation you may want to perform beforehand is a database vacuum, which will free up any unused space created from deleted objects. This will provide you with a more compact binary copy:

```
sqlite3 test.db VACUUM
cp test.db test.backup
```

As a general rule, binary backups are not as portable as SQL backups. On the whole, SQLite does have good backward compatibility and is binary compatible across all platforms for a given database format. However, for long-term backups, it is always a good idea to use SQL form. If size is an issue, SQL format (raw text) usually yields a good compression ratio.

Finally, if you've worked with other databases, "dropping" a database in SQLite, like binary backups, is a simple file operation: you simply delete the database file you wish to drop.

Getting Database File Information

As mentioned earlier, the primary means by which to obtain database information is using the sqlite_master view, which provides detailed information about all objects contained in a given database.

If you want information on the physical database structure, you can use a tool called SQLite Analyzer, which can be downloaded in binary form from the SQLite website. SQLite Analyzer provides detailed technical information about the on-disk structure of a SQLite database. This information includes a detailed breakdown of database, table, and index statistics for individual objects and in aggregate. It provides everything from database properties such as page size, number of tables, indexes, file size, and average page density (utilization) to detailed descriptions of individual database objects. Following the report is a detailed list of definitions explaining all terms used within the report. A partial output of sqlite_analyzer is as follows:

```
mike@linux tmp $ sqlite3_analyzer test.db
```

```
/** Disk-Space Utilization Report For test.db
*** As of 2005-May-07 20:26:23

Page size in bytes.................... 1024
Pages in the whole file (measured).... 3
Pages in the whole file (calculated).. 3
Pages that store data................. 3           100.0%
Pages on the freelist (per header).... 0             0.0%
Pages on the freelist (calculated).... 0             0.0%
Pages of auto-vacuum overhead......... 0             0.0%
Number of tables in the database...... 2
Number of indices..................... 1
Number of named indices............... 1
Automatically generated indices....... 0
Size of the file in bytes............. 3072
Bytes of user payload stored.......... 26            0.85%
```

```
*** Page counts for all tables with their indices ********************

TEST................................. 2           66.7%
SQLITE_MASTER........................ 1           33.3%

*** All tables and indices *******************************************

Percentage of total database.......... 100.0%
Number of entries.................... 11
Bytes of storage consumed............ 3072
Bytes of payload..................... 235          7.6%
Average payload per entry............ 21.36
Average unused bytes per entry........ 243.00
Maximum payload per entry............ 72
Entries that use overflow............ 0            0.0%
Primary pages used................... 3
Overflow pages used.................. 0
Total pages used..................... 3
Unused bytes on primary pages......... 2673        87.0%
Unused bytes on overflow pages........ 0
Unused bytes on all pages............ 2673        87.0%

*** Table TEST and all its indices ***********************************

Percentage of total database..........  66.7%
Number of entries.................... 8
Bytes of storage consumed............ 2048
Bytes of payload..................... 60           2.9%
Average payload per entry............ 7.50
Average unused bytes per entry........ 243.00
Maximum payload per entry............ 10
Entries that use overflow............ 0            0.0%
Primary pages used................... 2
Overflow pages used.................. 0
Total pages used..................... 2
Unused bytes on primary pages......... 1944        94.9%
Unused bytes on overflow pages........ 0
Unused bytes on all pages............ 1944        94.9%
```

SQLite Analyzer is provided in binary form on the SQLite website for Linux and Windows. On POSIX platforms, or with MinGW, SQLite Analyzer can be built from the source using the Unix makefile provided. From the build directory, issue the command

```
make sqlite3_analyzer
```

You must, however, have Tcl support configured in the build settings as SQLite Analyzer uses the Tcl extension to perform most of its work. Refer to "Compiling SQLite from Source" for more information.

Other SQLite Tools

There are many other open source and commercial programs available with which to work with SQLite. Good graphical, cross-platform tools include

- SQLite Database Browser (`http://sqlitebrowser.sourceforge.net`) is a program developed with Qt. With it users can manage databases, tables, and indexes, as well as import and export them. Users can interactively run SQL queries and inspect the results, as well as examine a log of all SQL commands issued. Users can also browse tables and modify their records.

- SQLite Control Center (`http://bobmanc.home.comcast.net/sqlitecc.html`) is a cross-platform program that uses the wxWindows C++ GUI framework. It does many of the same things as SQLite Database Browser, including general management of databases, tables, indexes, and triggers. Likewise, it also allows users to edit table data in a grid display and construct queries using a syntax-highlighting text editor.

- SQLiteManager (`www.sqlabs.net/sqlitemanager.php`) is a commercial software package designed for working with and administering SQLite. Users can manage database objects, execute queries, and save SQL, as well as create reports with flexible report templates.

These are just the cross-platform tools. Many more tools are available that can be used with PHP and other specific environments. You can find more information on such packages on the SQLite Wiki (`www.sqlite.org/cvstrac/wiki?p=SqliteTools`).

Summary

No matter what platform you work on, SQLite is quite easy to install and build. Windows and Linux users can obtain binaries directly from the SQLite website. Users of many other operating systems can also obtain binaries using their native—or even third-party—package systems.

The common way to work with SQLite across all platforms is using the SQLite command-line program (CLP). This program operates as both a command-line tool and an interactive shell. You can issue queries and do essential database administration tasks such as creating tables, indexes, and views as well as exporting and importing data. SQLite databases are contained in single operating system files, so doing things like binary backups are very simple—just copy the file. For long-term backups, however, it is always best to dump the database in SQL format, as this is portable across SQLite versions.

In the next few chapters, you will be using the CLP to explore SQL and the database aspects of SQLite. The next chapter provides a great deal of background to SQL. It is almost entirely theoretical. However, if you have previously used relational databases and SQL, you may find it very informative. It provides not only the basic theory underlying SQL but also a good bit of history as well. If you don't care for theory and want to dive right in, you may want to skip ahead to Chapter 4, where you will be able to put the CLP to immediate use.