Using

SQLite

O'REILLY®

Jay A. Kreibich

Table of Contents

CHAPTER 3

Building and Installing SQLite

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

SQLite Products

The SQLite project consists of four major products:

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

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

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

SQLite analyzer tool

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

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

Precompiled Distributions

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

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

Documentation Distribution

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

Source Distributions

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

The Amalgamation

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

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

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

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

Source Files

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

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

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

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

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

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

Source Downloads

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

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

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

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

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

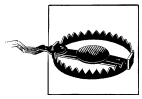
sqlite-source-3_x_x.zip
: The Windows source distribution. This is unsupported.

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

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

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

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



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

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

Building

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

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

Configure

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

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

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

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

Manually

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

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

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

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

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

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

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

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

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

Or, on Windows:

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

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

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

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

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

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

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

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

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

Build Customization

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

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

Build and Installation Options

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

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

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

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

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

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

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

An sqlite3 Primer

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Summary

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

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

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

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