Small. Fast. Reliable.
Choose any three.

How To Compile SQLite

SQLite is ANSI-C source code. It must be compiled into machine code before it is useful. This article is a guide to the various ways of compiling SQLite.

This article does not contain a step-by-step recipe for compiling SQLite. That would be difficult since each development situation is different. Rather, this article describes and illustrates the principals behind the compilation of SQLite. Typical compilation commands are provided as examples with the expectation that application developers can use these examples as guidance for developing their own custom compilation procedures. In other words, this article provides ideas and insights, not turnkey solutions.

Amalgamation Versus Individual Source Files

SQLite is built from over one hundred files of C code and script spread across multiple directories. The implementation of SQLite is pure ANSI-C, but many of the C-language source code files are either generated or transformed by auxiliary C programs and AWK, SED, and TCL scripts prior to being incorporated into the finished SQLite library. Building the necessary C programs and transforming and/or creating the C-language source code for SQLite is a complex process.

To simplify matters, SQLite is also available as a pre-packaged amalgamation source code file: **sqlite3.c**. The amalgamation is a single file of ANSI-C code that implements the entire SQLite library. The amalgamation is much easier to deal with. Everything is contained within a single code file, so it is easy to drop into the source tree of a larger C or C++ program. All the code generation and transformation steps have already been carried out so there are no auxiliary C programs to configure and compile and no scripts to run. And, because the entire library is contained in a single translation unit, compilers are able to do more advanced optimizations resulting in a 5% to 10% performance improvement. For these reasons, the amalgamation source file ("**sqlite3.c**") is recommended for all applications.

> *The use of the amalgamation is recommended for all applications.*

Building SQLite directly from individual source code files is certainly possible, but it is not recommended. For some specialized applications, it might be necessary to modify the build process in ways that cannot be done using just the prebuilt amalgamation source file downloaded from the website. For those situations, it is recommended that a customized amalgamation be built (as described below) and used. In other words, even if a project requires building SQLite beginning with individual source files, it is still recommended that an amalgamation source file be used as an intermediate step.

Compiling The Command-Line Interface

A build of the command-line interface requires three source files:

- **sqlite3.c**: The SQLite amalgamation source file
- **sqlite3.h**: The header files that accompanies sqlite3.c and defines the C-language interfaces to SQLite.
- **shell.c**: The command-line interface program itself. This is the C source code file that contains the definition of the **main()** routine and the loop that prompts for user input and passes that input into the SQLite database engine for processing.

All three of the above source files are contained in the amalgamation tarball available on the download page.

To build the CLI, simply put these three files in the same directory and compile them together. Using MSVC:

```
cl shell.c sqlite3.c -Fesqlite3.exe
```

On unix systems (or on Windows using cygwin or mingw+msys) the command typically looks something like this:

```
gcc shell.c sqlite3.c -lpthread -ldl
```

The pthreads library is needed to make SQLite threadsafe. But since the CLI is single threaded, we could instruct SQLite to build in a non-threadsafe mode and thereby omit the pthreads library:

```
gcc -DSQLITE_THREADSAFE=0 shell.c sqlite3.c -ldl
```

The -ldl library is needed to support dynamic loading, the sqlite3_load_extension() interface and the load_extension() SQL function. If these features are not required, then they can be omitted using SQLITE_OMIT_LOAD_EXTENSION compile-time option:

```
gcc -DSQLITE_THREADSAFE=0 -DSQLITE_OMIT_LOAD_EXTENSION shell.c sqlite3.c
```

One might want to provide other compile-time options such as -DSQLITE_ENABLE_FTS4 or -DSQLITE_ENABLE_FTS5 for full-text search, -DSQLITE_ENABLE_RTREE for the R*Tree search engine extension, -

-DSQLITE_ENABLE_JSON1 to include JSON SQL functions, or -DSQLITE_ENABLE_DBSTAT_VTAB for the dbstat virtual table. In order to see extra commentary in EXPLAIN listings, add the -DSQLITE_ENABLE_EXPLAIN_COMMENTS option. On unix systems, add -DHAVE_USLEEP=1 if the host machine supports the usleep() system call. Add -DHAVE_READLINE and the -lreadline and -lncurses libraries to get command-line editing support. One might also want to specify some compiler optimization switches. (The precompiled CLI available for download from the SQLite website uses "-Os".) There are countless possible variations here. A command to compile a full-featured shell might look something like this:

```
gcc -Os -I. -DSQLITE_THREADSAFE=0 -DSQLITE_ENABLE_FTS4 \
   -DSQLITE_ENABLE_FTS5 -DSQLITE_ENABLE_JSON1 \
   -DSQLITE_ENABLE_RTREE -DSQLITE_ENABLE_EXPLAIN_COMMENTS \
   -DHAVE_USLEEP -DHAVE_READLINE \
   shell.c sqlite3.c -ldl -lreadline -lncurses -o sqlite3
```

The key point is this: Building the CLI consists of compiling together two C-language files. The **shell.c** file contains the definition of the entry point and the user input loop and the SQLite amalgamation **sqlite3.c** contains the complete implementation of the SQLite library.

Compiling The TCL Interface

The TCL interface for SQLite is a small module that is added into the regular amalgamation. The result is a new amalgamated source file called "**tclsqlite3.c**". This single source file is all that is needed to generate a shared library that can be loaded into a standard tclsh or wish using the TCL load command, or to generate a standalone tclsh that comes with SQLite built in. A copy of the tcl amalgamation is included on the download page as a file in the TEA tarball.

To generate a TCL-loadable library for SQLite on Linux, the following command will suffice:

```
gcc -o libtclsqlite3.so -shared tclsqlite3.c -lpthread -ldl -ltcl
```

Building shared libraries for Mac OS X and Windows is not nearly so simple, unfortunately. For those platforms it is best to use the configure script and makefile that is included with the TEA tarball.

To generate a standalone tclsh that is statically linked with SQLite, use this compiler invocation:

```
gcc -DTCLSH=1 tclsqlite3.c -ltcl -lpthread -ldl -lz -lm
```

The trick here is the -DTCLSH=1 option. The TCL interface module for SQLite includes a **main()** procedure that initializes a TCL interpreter and enters a command-line loop when it is compiled with -DTCLSH=1. The command above works on both Linux and Mac OS X, though one may need to adjust the library options depending on the platform and which version of TCL one is linking against.

Building The Amalgamation

The versions of the SQLite amalgamation that are supplied on the download page are normally adequate for most users. However, some projects may want or need to build their own amalgamations. A common reason for building a custom amalgamation is in order to use certain compile-time options to customize the SQLite library. Recall that the SQLite amalgamation contains a lot of C-code that is generated by auxiliary programs and scripts. Many of the compile-time options effect this generated code and must be supplied to the code generators before the amalgamation is assembled. The set of compile-time options that must be passed into the code generators can vary from one release of SQLite to the next, but at the time of this writing (circa SQLite 3.6.20, 2009-11-04) the set of options that must be known by the code generators includes:

- SQLITE_ENABLE_UPDATE_DELETE_LIMIT
- SQLITE_OMIT_ALTERTABLE
- SQLITE_OMIT_ANALYZE
- SQLITE_OMIT_ATTACH
- SQLITE_OMIT_AUTOINCREMENT
- SQLITE_OMIT_CAST
- SQLITE_OMIT_COMPOUND_SELECT
- SQLITE_OMIT_EXPLAIN
- SQLITE_OMIT_FOREIGN_KEY
- SQLITE_OMIT_PRAGMA
- SQLITE_OMIT_REINDEX
- SQLITE_OMIT_SUBQUERY
- SQLITE_OMIT_TEMPDB
- SQLITE_OMIT_TRIGGER
- SQLITE_OMIT_VACUUM
- SQLITE_OMIT_VIEW
- SQLITE_OMIT_VIRTUALTABLE

To build a custom amalgamation, first download the original individual source files onto a unix or unix-like development platform. Be sure to get the original source files not the "preprocessed source files". One can obtain the complete set of original source files either from the download page or directly from the configuration management system.

Suppose the SQLite source tree is stored in a directory named "sqlite". Plan to construct the amalgamation in a parallel directory named (for example) "bld". First construct an appropriate Makefile by either running the configure script at the top of the SQLite source tree, or by making a copy of one of the template Makefiles at the top of the source tree. Then hand edit this Makefile to include the desired compile-time options. Finally run:

```
make sqlite3.c
```

Or on Windows with MSVC:

```
nmake /f Makefile.msc sqlite3.c
```

The "sqlite3.c" make target will automatically construct the regular "**sqlite3.c**" amalgamation source file, its header file "**sqlite3.h**", and the "**tclsqlite3.c**" amalgamation source file that includes the TCL interface. Afterwards, the needed files can be copied into project directories and compiled according to the procedures outlined above.

Building A Windows DLL

To build a DLL of SQLite for use in Windows, first acquire the appropriate amalgamated source code files, sqlite3.c and sqlite3.h. These can either be downloaded from the SQLite website or custom generated from sources as shown above.

With source code files in the working directory, a DLL can be generated using MSVC with the following command:

```
cl sqlite3.c -link -dll -out:sqlite3.dll
```

The above command should be run from the MSVC Native Tools Command Prompt. If you have MSVC installed on your machine, you probably have multiple versions of this Command Prompt, for native builds for x86 and x64, and possibly also for cross-compiling to ARM. Use the appropriate Command Prompt depending on the desired DLL.

If using the MinGW compiler, the command-line is this:

```
gcc -shared sqlite3.c -o sqlite3.dll
```

Note that MinGW generates 32-bit DLLs only. There is a separate MinGW64 project that can be used to generate 64-bit DLLs. Presumably the command-line syntax is similar. Also note that recent versions of MSVC generate DLLs that will not work on WinXP and earlier versions of Windows. So for maximum compatibility of your generated DLL, MinGW is recommended. A good rule-of-thumb is to generate 32-bit DLLs using MinGW and 64-bit DLLs using MSVC.

In most cases, you will want to supplement the basic commands above with compile-time options appropriate for your application. Commonly used compile-time options include:

- **-Os** - Optimize for size. Make the DLL as small as possible.
- **-O2** - Optimize for speed. This will make the DLL larger by unrolling loops and inlining functions.
- **-DSQLITE_ENABLE_FTS4** - Include the full-text search engine code in SQLite.
- **-DSQLITE_ENABLE_RTREE** - Include the R-Tree extension.

- **-DSQLITE_ENABLE_COLUMN_METADATA** - This enables some extra APIs that are required by some common systems, including Ruby-on-Rails.