An Introduction To The SQLite C/C++ Interface

▶ Table Of Contents

1. Summary

The following two objects and eight methods comprise the essential elements of the SQLite interface:

- **sqlite3** → The database connection object. Created by sqlite3_open() and destroyed by sqlite3_close().
- **sqlite3_stmt** → The prepared statement object. Created by sqlite3_prepare() and destroyed by sqlite3_finalize().
- **sqlite3_open()** → Open a connection to a new or existing SQLite database. The constructor for sqlite3.
- **sqlite3_prepare()** → Compile SQL text into byte-code that will do the work of querying or updating the database. The constructor for sqlite3_stmt.
- **sqlite3_bind()** → Store application data into parameters of the original SQL.
- **sqlite3_step()** → Advance an sqlite3_stmt to the next result row or to completion.
- **sqlite3_column()** → Column values in the current result row for an sqlite3_stmt.
- **sqlite3_finalize()** → Destructor for sqlite3_stmt.
- **sqlite3_close()** → Destructor for sqlite3.
- **sqlite3_exec()** → A wrapper function that does sqlite3_prepare(), sqlite3_step(), sqlite3_column(), and sqlite3_finalize() for a string of one or more SQL statements.

2. Introduction

SQLite has more than 225 APIs. However, most of the APIs are optional and very specialized and can be ignored by beginners. The core API is small, simple, and easy to learn. This article summarizes the core API.

A separate document, The SQLite C/C++ Interface, provides detailed specifications for all C/C++ APIs for SQLite. Once the reader understands the basic principles of operation for SQLite, that document should be used as a reference guide. This article is intended as introduction only and is neither a complete nor authoritative reference for the SQLite API.

3. Core Objects And Interfaces

The principal task of an SQL database engine is to evaluate SQL statements of SQL. To accomplish this, the developer needs two objects:

- The database connection object: sqlite3
- The prepared statement object: sqlite3_stmt

Strictly speaking, the prepared statement object is not required since the convenience wrapper interfaces, sqlite3_exec or sqlite3_get_table, can be used and these convenience wrappers encapsulate and hide the prepared statement object. Nevertheless, an understanding of prepared statements is needed to make full use of SQLite.

The database connection and prepared statement objects are controlled by a small set of C/C++ interface routine listed below.

- sqlite3_open()
- sqlite3_prepare()
- sqlite3_step()
- sqlite3_column()
- sqlite3_finalize()
- sqlite3_close()

Note that the list of routines above is conceptual rather than actual. Many of these routines come in multiple versions. For example, the list above shows a single routine named sqlite3_open() when in fact there are three separate routines that accomplish the same thing in slightly different ways: sqlite3_open(), sqlite3_open16() and sqlite3_open_v2(). The list mentions sqlite3_column() when in fact no such routine exists. The "sqlite3_column()" shown in the list is a placeholder for an entire family of routines that extra column data in various datatypes.

Here is a summary of what the core interfaces do:

- **sqlite3_open()**

  This routine opens a connection to an SQLite database file and returns a database connection object. This is often the first SQLite API call that an application makes and is a prerequisite for most other SQLite APIs. Many SQLite interfaces require a pointer to the database connection object as their first parameter and can be thought of as methods on the database connection object. This routine is the constructor for the database connection object.

- **sqlite3_prepare()**

This routine converts SQL text into a prepared statement object and returns a pointer to that object. This interface requires a database connection pointer created by a prior call to sqlite3_open() and a text string containing the SQL statement to be prepared. This API does not actually evaluate the SQL statement. It merely prepares the SQL statement for evaluation.

Think of each SQL statement as a small computer program. The purpose of sqlite3_prepare() is to compile that program into object code. The prepared statement is the object code. The sqlite3_step() interface then runs the object code to get a result.

New applications should always invoke sqlite3_prepare_v2() instead of sqlite3_prepare(). The older sqlite3_prepare() is retained for backwards compatibility. But sqlite3_prepare_v2() provides a much better interface.

- **sqlite3_step()**

  This routine is used to evaluate a prepared statement that has been previously created by the sqlite3_prepare() interface. The statement is evaluated up to the point where the first row of results are available. To advance to the second row of results, invoke sqlite3_step() again. Continue invoking sqlite3_step() until the statement is complete. Statements that do not return results (ex: INSERT, UPDATE, or DELETE statements) run to completion on a single call to sqlite3_step().

- **sqlite3_column()**

  This routine returns a single column from the current row of a result set for a prepared statement that is being evaluated by sqlite3_step(). Each time sqlite3_step() stops with a new result set row, this routine can be called multiple times to find the values of all columns in that row.

  As noted above, there really is no such thing as a "sqlite3_column()" function in the SQLite API. Instead, what we here call "sqlite3_column()" is a place-holder for an entire family of functions that return a value from the result set in various data types. There are also routines in this family that return the size of the result (if it is a string or BLOB) and the number of columns in the result set.

  - sqlite3_column_blob()
  - sqlite3_column_bytes()
  - sqlite3_column_bytes16()
  - sqlite3_column_count()
  - sqlite3_column_double()
  - sqlite3_column_int()
  - sqlite3_column_int64()
  - sqlite3_column_text()
  - sqlite3_column_text16()
  - sqlite3_column_type()
  - sqlite3_column_value()

- **sqlite3_finalize()**

  This routine destroys a prepared statement created by a prior call to sqlite3_prepare(). Every prepared statement must be destroyed using a call to this routine in order to avoid memory leaks.

- **sqlite3_close()**

  This routine closes a database connection previously opened by a call to sqlite3_open(). All prepared statements associated with the connection should be finalized prior to closing the connection.

4. Typical Usage Of Core Routines And Objects

An application will typically use sqlite3_open() to create a single database connection during initialization. Note that sqlite3_open() can be used to either open existing database files or to create and open new database files. While many applications use only a single database connection, there is no reason why an application cannot call sqlite3_open() multiple times in order to open multiple database connections - either to the same database or to different databases. Sometimes a multi-threaded application will create separate database connections for each thread. Note that a single database connection can access two or more databases using the ATTACH SQL command, so it is not necessary to have a separate database connection for each database file.

Many applications destroy their database connections using calls to sqlite3_close() at shutdown. Or, for example, an application that uses SQLite as its application file format might open database connections in response to a File/Open menu action and then destroy the corresponding database connection in response to the File/Close menu.

To run an SQL statement, the application follows these steps:

1. Create a prepared statement using sqlite3_prepare().
2. Evaluate the prepared statement by calling sqlite3_step() one or more times.
3. For queries, extract results by calling sqlite3_column() in between two calls to sqlite3_step().
4. Destroy the prepared statement using sqlite3_finalize().

The foregoing is all one really needs to know in order to use SQLite effectively. All the rest is optimization and detail.

5. Convenience Wrappers Around Core Routines

The sqlite3_exec() interface is a convenience wrapper that carries out all four of the above steps with a single function call. A callback function passed into sqlite3_exec() is used to process each row of the result set. The sqlite3_get_table() is another convenience wrapper that does all four of the above steps. The sqlite3_get_table() interface differs from sqlite3_exec() in that it stores the results of queries in heap memory rather than invoking a callback.

It is important to realize that neither sqlite3_exec() nor sqlite3_get_table() do anything that cannot be accomplished using the core routines. In fact, these wrappers are implemented purely in terms of the core routines.

6. Binding Parameters and Reusing Prepared Statements

In prior discussion, it was assumed that each SQL statement is prepared once, evaluated, then destroyed. However, SQLite allows the same prepared statement to be evaluated multiple times. This is accomplished using the following routines:

- sqlite3_reset()
- sqlite3_bind()

After a prepared statement has been evaluated by one or more calls to sqlite3_step(), it can be reset in order to be evaluated again by a call to sqlite3_reset(). Think of sqlite3_reset() as rewinding the prepared statement program back to the beginning. Using sqlite3_reset() on an existing prepared statement rather than creating a new prepared statement avoids unnecessary calls to sqlite3_prepare(). For many SQL statements, the time needed to run sqlite3_prepare() equals or exceeds the time needed by sqlite3_step(). So avoiding calls to sqlite3_prepare() can give a significant performance improvement.

It is not commonly useful to evaluate the *exact* same SQL statement more than once. More often, one wants to evaluate similar statements. For example, you might want to evaluate an INSERT statement multiple times with different values. Or you might want to evaluate the same query multiple times using a different key in the WHERE clause. To accommodate this, SQLite allows SQL statements to contain parameters which are "bound" to values prior to being evaluated. These values can later be changed and the same prepared statement can be evaluated a second time using the new values.

SQLite allows a parameter wherever a string literal, numeric constant, or NULL is allowed. (Parameters may not be used for column or table names.) A parameter takes one of the following forms:

- **?**
- **?***NNN*
- **:***AAA*
- **$***AAA*
- **@***AAA*

In the examples above, *NNN* is an integer value and *AAA* is an identifier. A parameter initially has a value of NULL. Prior to calling sqlite3_step() for the first time or immediately after sqlite3_reset(), the application can invoke the sqlite3_bind() interfaces to attach values to the parameters. Each call to sqlite3_bind() overrides prior bindings on the same parameter.

An application is allowed to prepare multiple SQL statements in advance and evaluate them as needed. There is no arbitrary limit to the number of outstanding prepared statements. Some applications call sqlite3_prepare() multiple times at start-up to create all of the prepared statements they will ever need. Other applications keep a cache of the most recently used prepared statements and then reuse prepared statements out of the cache when available. Another approach is to only reuse prepared statements when they are inside of a loop.

7. Configuring SQLite

The default configuration for SQLite works great for most applications. But sometimes developers want to tweak the setup to try to squeeze out a little more performance, or take advantage of some obscure feature.

The sqlite3_config() interface is used to make global, process-wide configuration changes for SQLite. The sqlite3_config() interface must be called before any database connections are created. The sqlite3_config() interface allows the programmer to do things like:

- Adjust how SQLite does memory allocation, including setting up alternative memory allocators appropriate for safety-critical real-time embedded systems and application-defined memory allocators.
- Set up a process-wide error log.
- Specify an application-defined page cache.
- Adjust the use of mutexes so that they are appropriate for various threading models, or substitute an application-defined mutex system.

After process-wide configuration is complete and database connections have been created, individual database connections can be configured using calls to sqlite3_limit() and sqlite3_db_config().

8. Extending SQLite

SQLite includes interfaces that can be used to extend its functionality. Such routines include:

- sqlite3_create_collation()
- sqlite3_create_function()
- sqlite3_create_module()
- sqlite3_vfs_register()

The sqlite3_create_collation() interface is used to create new collating sequences for sorting text. The sqlite3_create_module() interface is used to register new virtual table implementations. The sqlite3_vfs_register() interface creates new VFSes.

The sqlite3_create_function() interface creates new SQL functions - either scalar or aggregate. The new function implementation typically makes use of the following additional interfaces:

- sqlite3_aggregate_context()
- sqlite3_result()
- sqlite3_user_data()
- sqlite3_value()

All of the built-in SQL functions of SQLite are created using exactly these same interfaces. Refer to the SQLite source code, and in particular the date.c and func.c source files for examples.

Shared libraries or DLLs can be used as loadable extensions to SQLite.

9. Other Interfaces

This article only mentions the most important and most commonly used SQLite interfaces. The SQLite library includes many other APIs implementing useful features that are not described here. A complete list of functions that form the SQLite application programming interface is found at the C/C++ Interface Specification. Refer to that document for complete and authoritative information about all SQLite interfaces.