

# 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.