Using

SQLite

O'REILLY®

Jay A. Kreibich

Table of Contents

CHAPTER 7

C Programming Interface

The `sqlite3` command-line utility is designed to provide an interactive interface for end users. It is extremely useful to design and test out SQL queries, debug database files, and experiment with new features of SQLite, but it was never meant to interface with other applications. While the command-line utility can be used for very basic scripting and automated tasks, if you want to write an application that utilizes the SQLite library in any serious manner, it is expected that you'll use a programming interface.

The native SQLite programming interface is in C, and that is the interface this chapter will cover. If you're working in something else, there are wrappers and extensions available for many other languages, including most popular scripting languages. With the exception of the Tcl interface, all of these wrappers are provided by third parties and are not part of the SQLite product. For more information on language wrappers, see "Scripting Languages and Other Interfaces" on page 172.

Using the C API allows your application to interface directly with the SQLite library and the database engine. You can link a static or dynamic build of the SQLite library into your application, or simply include the amalgamation source file in your application's build process. The best choice depends on your specific situation. See "Build and Installation Options" on page 23 for more details.

The C API is fairly extensive, and provides full access to all of SQLite's features. In fact, the `sqlite3` command-line utility is written using the public C API. This chapter covers the core features of the API, while the following chapters cover more advanced features.

API Overview

Even when using the programming interface, the primary way of interacting with your data is to issue SQL commands to the database engine. This chapter focuses on the core of the API that is used to convey SQL command strings to the database engine. It is important to understand that there are no public functions to walk the internal structure of a table or, for example, access the tree structure of an index. You must use SQL to query data from the database. In order to be successful with the SQLite API,

you not only need to understand the C API, but you also need to know enough SQL to form meaningful and efficient queries.

Structure

The C API for SQLite 3 includes a dozen-plus data structures, a fair number of constants, and well over one hundred different function calls. While the API is somewhat large, using it doesn't have to be complex. A fair number of the functions are highly specialized and infrequently used by most developers. Many of the remaining functions are simple variations of the same basic operation. For example, there are a dozen variations on the `sqlite3_value_xxx()` function, such as `sqlite3_value_int()`, `sqlite3_value_double()`, and `sqlite3_value_text()`. All of these functions perform the same basic operation and can be considered simple type variations of the same basic interface.

When referring to a whole category of functions, either in text or in pseudo code, I'll simply refer to them as the `sqlite3_value_xxx()` functions. Much of the SQLite documentation refers to them as `sqlite3_value_*()`, but I prefer to use the `xxx` notation to avoid any confusion with pointers. There are no actual SQLite3 functions with the letter sequence `xxx` in the name.

All public API function calls and datatypes have the prefix `sqlite3_`, indicating they are part of version 3.x of the SQLite product. Most of the constants, such as error codes, use the prefix `SQLITE_`. The design and API differences between SQLite 2.x and 3.x were significant enough to warrant a complete change of all API names and structures. The depth of these changes required anyone upgrading from SQLite 2 to SQLite 3 to modify their application, so changing the names of the API functions only helped keep the names distinct and keep any version questions clear. The distinct names also allowed applications that were in transition to link to both libraries at the same time.

In addition to the `sqlite3_` prefix, public function calls can be identified by the use of lowercase letters and underscores in their names. Private functions use run-together capitalized words (also known as CamelCase). For example, `sqlite3_create_function()` is a public API function (used to register a user-defined SQL function), while `sqlite3CreateFunc()` is an internal function that should never be called directly. Internal functions are not in the public header file, are not documented, and are subject to change at any time.

The stability of the public interface is extremely important to the SQLite development team. An existing API function call will not be altered once it has been made public. The only possible exceptions are brand new interfaces that are marked *experimental*, and even experimental interfaces tend to become fairly solid after a few releases.

If a revised version of a function call is needed, the newer function will generally be introduced with the suffix `_v2`. For example, when a more flexible version of the existing `sqlite3_open()` function was introduced, the old version of the function was retained as is and the new, improved `sqlite3_open_v2()` was introduced. Although no `_v3` (or higher) functions currently exist, it is possible they may be introduced in the future.

By adding a new function, rather than modifying the parameters of an existing function, new code could take advantage of the newer features, while existing, unmodified code could continue to link against updated versions of the SQLite library. The use of a different function name also means that if a newer application is ever accidentally linked against an older version of the library, the result will be a link error rather than a program crash, making it much easier to track down and resolve the problem.

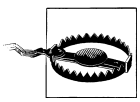
Strings and Unicode

There are a number of API functions that have a `16` variant. For instance, both an `sqlite3_column_text()` function and an `sqlite3_column_text16()` function are available. The first requests a text value in UTF-8 format, while the second will request a text value in UTF-16.

All of the strings in an SQLite database file are stored using the same encoding. SQLite database files support the UTF-8, UTF-16LE, and UTF-16BE encodings. A database's encoding is determined when the database is created.

Regardless of the database, you can insert or request text values in either UTF-8 or UTF-16. SQLite will automatically convert text values between the database encoding and the API encoding. The UTF-16 encoding passed by the `16` APIs will always be in the machine's native byte order. UTF-16 buffers use a `void*` C data type. The `wchar_t` data type is not used, as its size is not fixed, and not all platforms define a 16-bit type.

Most of the string- and text-related functions have some type of length parameter. SQLite does not assume input text values are null-terminated, so explicit lengths are often required. These lengths are always given in bytes, not characters, regardless of the string encoding.



All string lengths are given in *bytes*, not *characters*, even if the string uses a multi-byte encoding such as UTF-16.

This difference is important to keep in mind when using international strings.

Error Codes

SQLite follows the convention of returning integer error codes in any situation when there is a chance of failure. If data needs to be passed back to the function caller, it is returned through a reference parameter.

In all cases, if a function succeeds, it will return the constant `SQLITE_OK`, which happens to have the value zero. If something went wrong, API functions will return one of the standard error codes to indicate the nature of the error.

More recently, a set of extended error codes were introduced. These provide a more specific indication of what went wrong. However, to keep things backwards compatible, these extended codes are only available when you activate them.

The situation is complex enough to warrant its own discussion later in the chapter. It will be much easier to explain the different error codes once you've had a chance to see how the API works. See "Result Codes and Error Codes" on page 146 for more details.

I also have to give the standard "do as I say, not as I do" caveat about properly checking error codes and return results. The example code in this chapter and elsewhere in this book tends to have extremely terse (as in, almost none at all) error checking. This is done to keep the examples short and clear. Needless to say, this isn't the best approach for production code. When working with your own code, do the right thing and check your error codes.

Structures and Allocations

Although the native SQLite API is often referred to as a C/C++ API, technically the interface is only available in C. As mentioned in "Building" on page 21, the SQLite source code is strictly C based, and as such can only be compiled with a C compiler. Once compiled, the library can be easily linked to, and called from, both C and C++ code, as well as any other language that follows the C linking conventions for your platform.

Although the API is written in C, it has a distinct object-like flavor. Most of the program state is held in a series of opaque data structures that act like objects. The most common data structures are database connections and prepared statements. You should never directly access the fields of these data structures. Instead, functions are provided to create, destroy, and manipulate these structures in much the same way that object methods are used to manipulate object instances. This results in an API design that has similar feelings to an object-oriented design. In fact, if you download one of the third-party C++ wrappers, you'll notice that the wrappers tend to be rather thin, owing most of their structure to the underlying C functions and the data structures.

It is important that you allow SQLite to allocate and manage its own data structures. The design of the API means that you should never manually allocate one of these structures, nor should you put these structures on the stack. The API provides calls to

internally allocate the proper data structures, initialize them, and return them. Similarly, for every function that allocates a data structure, there is some function that is used to clean it up and release it. As with memory management, you need to be sure these calls are balanced, and that every data structure that is created is eventually released.

More Info

The core of the SQLite API focuses on opening database connections, preparing SQL statements, binding parameter values, executing statements, and finally stepping through the results. These procedures are the focus of this chapter.

There are also interfaces to create your own SQL functions, load dynamic modules, and create code-driven virtual tables. We'll be covering some of these more advanced interfaces in other chapters.

Beyond that, there are a fair number of management and customization functions. Not all of these are covered in the main part of the book, but a reference for the full API can be found in Appendix G. If the description of a function leaves you with additional questions, be sure to check that appendix for more specific details.

Library Initialization

Before an application can use the SQLite library to do anything, the library must first be initialized. This process allocates some standard resources and sets up any OS-specific data structures. By default, most major API function calls will automatically initialize the library, if it has not already been initialized. It is considered a good practice to manually initialize the library, however.

`int sqlite3_initialize( )`
: Initializes the SQLite library. This function should be called prior to any other function in the SQLite API. Calling this function after the library has already been initialized is harmless. This function can be called after a shutdown to reinitialize the library. A return value of `SQLITE_OK` indicates success.

When an application is finished using the SQLite library, the library should be shut down.

`int sqlite3_shutdown( )`
: Releases any resources allocated by `sqlite3_initialize()`. Calling this function before the library has been initialized or after the library has already been shut down is harmless. A return value of `SQLITE_OK` indicates success.

Because of the automatic initialization features, many applications never call either of these functions. Rather, they call `sqlite3_open()`, or one of the other primary functions, and depend on the library to automatically initialize itself. In most cases this is safe enough, but for maximum compatibility it is best to call these functions explicitly.

Database Connections

Before we can prepare or execute SQL statements, we must first establish a database connection. Most often this is done by opening or creating an SQLite3 database file. When you are done with the database connection, it must be closed. This verifies that there are no outstanding statements or allocated resources before closing the database file.

Opening

Database connections are allocated and established with one of the `sqlite3_open_xxx()` commands. These pass back a database connection in the form of an `sqlite3` data structure. There are three variants:

```
int sqlite3_open(   const char *filename, sqlite3 **db_ptr )
int sqlite3_open16( const void *filename, sqlite3 **db_ptr )
```

Opens a database file and allocates an `sqlite3` data structure. The first parameter is the filename of the database file you wish to open, given as a null-terminated string. The second parameter is a reference to an `sqlite3` pointer, and is used to pass back the new connection. If possible, the database will be opened read/write. If not, it will be opened read-only. If the given database file does not exist, it will be created.

The first variant assumes that the database filename is encoded in UTF-8, while the second assumes that the database filename is encoded in UTF-16.

```
int sqlite3_open_v2( const char *filename, sqlite3 **db_ptr,
                     int flags, const char *vfs_name )
```

The `_v2` variant offers more control over how the database file is created and opened. The first two parameters are the same. The filename is assumed to be in UTF-8. There is no UTF-16 variant of this function.

A third parameter is a set of bit-field flags. These flags allow you to specify if SQLite should attempt to open the database read/write (`SQLITE_OPEN_READWRITE`), or read-only (`SQLITE_OPEN_READONLY`). If you ask for read/write access but only read-only access is available, the database will be opened in read-only mode.

This variant of open will not create a new file for an unknown filename unless you explicitly allow it using the `SQLITE_OPEN_CREATE` flag. This only works if the database is being opened in read/write mode.

There are also a number of other flags dealing with thread and cache management. See `sqlite3_open()` in Appendix G for more details. The standard version of open is equivalent to the flag values of (`SQLITE_READWRITE | SQLITE_CREATE`).

The final parameter allows you to name a VFS (Virtual File System) module to use with this database connection. The VFS system acts as an abstraction layer between the SQLite library and any underlying storage system (such as a filesystem). In nearly all cases, you will want to use the default VFS module and can simply pass in a NULL pointer.

For new code, it is recommended that you use the call `sqlite3_open_v2()`. The newer call allows more control over how the database is opened and processed.

The use of the double pointer may be a bit confusing at first, but the idea behind it is simple enough. The pointer-to-a-pointer is really nothing more than a pointer that is passed by reference. This allows the function call to modify the pointer that is passed in. For example:

```
sqlite3    *db = NULL;
rc = sqlite3_open_v2( "database.sqlite3", &db, SQLITE_OPEN_READWRITE, NULL );
/* hopefully, db now points to a valid sqlite3 structure */
```

Note that `db` is an `sqlite3` pointer (`sqlite3*`), not an actual `sqlite3` structure. When we call `sqlite3_open_xxx()` and pass in the pointer reference, the open function will allocate a new `sqlite3` data structure, initialize it, and set our pointer to point to it.

This approach, including the use of a pointer reference, is a common theme in the SQLite APIs that are used to create or initialize something. They all basically work the same way and, once you get the hang of them, they are pretty straightforward and easy to use.

There is no standard file extension for an SQLite3 database file, although *.sqlite3*, *.db*, and *.db3* are popular choices. The extension *.sdb* should be avoided, as this extension has special meaning on some Microsoft Windows platforms, and may suffer from significantly slower I/O performance.

The string encoding used by a database file is determined by the function that is used to create the file. Using `sqlite3_open()` or `sqlite3_open_v2()` will result in a database with the default UTF-8 encoding. If `sqlite3_open16()` is used to create a database, the default string encoding will be UTF-16 in the native byte order of the machine. You can override the default string encoding with the SQL command `PRAGMA encoding`. See `encoding` in Appendix F for more details.

Special Cases

In addition to recognizing standard filenames, SQLite recognizes a few specialized filename strings. If the given filename is a NULL pointer or an empty string (`""`), then an anonymous, temporary, on-disk database is created. An anonymous database can only be accessed through the database connection that created it. Each call will create a new, unique database instance. Like all temporary items, this database will be destroyed when the connection is closed.

If the filename `:memory:` is used, then a temporary, in-memory database is created. In-memory databases live in the database cache and have no backing store. This style of database is extremely fast, but requires sufficient memory to hold the entire database image in memory. As with anonymous databases, each open call will create a new, unique, in-memory database, making it impossible for more than one database connection to access a given in-memory database.

In-memory databases make great structured caches. It is not possible to directly image an in-memory database to disk, but you can copy the contents of an in-memory database to disk (or disk to memory) using the database backup API. See the section `sqlite3_backup_init()` in Appendix G for more details.

Closing

To close and release a database connection, call `sqlite3_close()`.

`int sqlite3_close( sqlite3 *db )`
: Closes a database connection and releases any associated data structures. All temporary items associated with this connection will be deleted. In order to succeed, all prepared statements associated with this database connection must be finalized. See "Reset and Finalize" on page 130 for more details.

Any pointer returned by a call to `sqlite3_open_xxx()`, including a NULL pointer, can be passed to `sqlite3_close()`. This function verifies there are no outstanding changes to the database, then closes the file and frees the `sqlite3` data structure. If the database still has nonfinalized statements, the `SQLITE_BUSY` error will be returned. In that case, you need to correct the problem and call `sqlite3_close()` again.

In most cases, `sqlite3_open_xxx()` will return a pointer to an `sqlite3` structure, even when the return code indicates a problem. This allows the caller to retrieve an error message with `sqlite3_errmsg()`. (See "Result Codes and Error Codes" on page 146.) In these situations, you must still call `sqlite3_close()` to free the `sqlite3` structure.

Example

Here is the outline of a program that opens a database, performs some operations, and then closes it. Most of the other examples in this chapter will build from this example by inserting code into the middle:

```
#include "sqlite3.h"
#include <stdlib.h>

int main( int argc, char **argv )
{
    char            *file = ""; /* default to temp db */
    sqlite3         *db = NULL;
    int             rc = 0;
```

```
    if ( argc > 1 )
        file = argv[1];

    sqlite3_initialize( );
    rc = sqlite3_open_v2( file, &db, SQLITE_OPEN_READWRITE |
                                     SQLITE_OPEN_CREATE, NULL );
    if ( rc != SQLITE_OK) {
        sqlite3_close( db );
        exit( -1 );
    }

    /*  perform database operations  */

    sqlite3_close( db );
    sqlite3_shutdown( );
}
```

The default filename is an empty string. If passed to `sqlite3_open_xxx()`, this will result in a temporary, on-disk database that will be deleted as soon as the database connection is closed. If at least one argument is given, the first argument will be used as a filename. If the database does not exist, it will be created and then opened for read/write access. It is then immediately closed.

If this example is run using a new filename, it will not create a valid database file. The SQLite library delays writing the database header until some actual data operation is performed. This "lazy" initialization gives an application the chance to adjust any relevant pragmas, such as the text encoding, page size, and database file format, before the database file is fully created.

Prepared Statements

Once a database connection is established, we can start to execute SQL commands. This is normally done by preparing and stepping through statements. Statements are held in `sqlite3_stmt` data structures.

Statement Life Cycle

The life cycle of a prepared statement is a bit complex. Unlike database connections, which are typically opened, used for some period of time, and then closed, a statement can be in a number of different states. A statement might be prepared, but not run, or it might be in the middle of processing. Once a statement has run to completion, it can be reset and re-executed multiple times before eventually being finalized and released.

The life cycle of a typical `sqlite3_stmt` looks something like this (in pseudo-code):

```
/* create a statement from an SQL string */
sqlite3_stmt *stmt = NULL;
sqlite3_prepare_v2( db, sql_str, sql_str_len, &stmt, NULL );

/* use the statement as many times as required */
while( ... )
{
    /* bind any parameter values */
    sqlite3_bind_xxx( stmt, param_idx, param_value... );
    ...

    /* execute statement and step over each row of the result set */
    while ( sqlite3_step( stmt ) == SQLITE_ROW )
    {
        /* extract column values from the current result row */
        col_val = sqlite3_column_xxx( stmt, col_index );
        ...
    }

    /* reset the statement so it may be used again */
    sqlite3_reset( stmt );
    sqlite3_clear_bindings( stmt );  /* optional */
}

/* destroy and release the statement */
sqlite3_finalize( stmt );
stmt = NULL;
```

The prepare process converts an SQL command string into a prepared statement. That statement can then have values bound to any statement parameters. The statement is then executed, or "stepped through." In the case of a query, each step will make a new results row available for processing. The column values of the current row can then be extracted and processed. The statement is stepped through, row by row, until no more rows are available.

The statement can then be reset, allowing it to be re-executed with a new set of bindings. Preparing a statement can be somewhat costly, so it is a common practice to reuse statements as much as possible. Finally, when the statement is no longer in use, the `sqlite3_stmt` data structure can be finalized. This releases any internal resources and frees the `sqlite3_stmt` data structure, effectively deleting the statement.

Prepare

To convert an SQL command string into a prepared statement, use one of the `sqlite3_prepare_xxx()` functions:

```
int sqlite3_prepare(   sqlite3 *db, const char *sql_str, int sql_str_len,
                       sqlite3_stmt **stmt, const char **tail )
int sqlite3_prepare16( sqlite3 *db, const void *sql_str, int sql_str_len,
                       sqlite3_stmt **stmt, const void **tail )
```

It is strongly recommended that all new developments use the `_v2` version of these functions.

```
int sqlite3_prepare_v2(   sqlite3 *db, const char *sql_str, int sql_str_len,
                          sqlite3_stmt **stmt, const char **tail )
int sqlite3_prepare16_v2( sqlite3 *db, const void *sql_str, int sql_str_len,
                          sqlite3_stmt **stmt, const void **tail )
```

Converts an SQL command string into a prepared statement. The first parameter is a database connection. The second parameter is an SQL command encoded in a UTF-8 or UTF-16 string. The third parameter indicates the length of the command string in bytes. The fourth parameter is a reference to a statement pointer. This is used to pass back a pointer to the new `sqlite3_stmt` structure.

The fifth parameter is a reference to a string (`char` pointer). If the command string contains multiple SQL statements and this parameter is non-NULL, the pointer will be set to the start of the next statement in the command string.

These `_v2` calls take the exact same parameters as the original versions, but the internal representation of the `sqlite3_stmt` structure that is created is somewhat different. This enables some extended and automatic error handling. These differences are discussed later in "Result Codes and Error Codes" on page 146.

If the length parameter is negative, the length will be automatically computed by the prepare call. This requires that the command string be properly null-terminated. If the length is positive, it represents the maximum number of bytes that will be parsed. For optimal performance, provide a null-terminated string and pass a valid length value that includes the null-termination character. If the SQL command string passed to `sqlite3_prepare_xxx()` consists of only a single SQL statement, there is no need to terminate it with a semicolon.

Once a statement has been prepared, but before it is executed, you can bind parameter values to the statement. Statement parameters allow you to insert a special token into the SQL command string that represents an unspecified literal value. You can then bind specific values to the parameter tokens before the statement is executed. After execution, the statement can be reset and new parameter values can be assigned. This allows you to prepare a statement once and then re-execute it multiple times with different parameter values. This is commonly used with commands, such as `INSERT`, that have a common structure but are repeatedly executed with different values.

Parameter binding is a somewhat in-depth topic, so we'll get back to that in the next section. See "Bound Parameters" on page 133 for more details.

Step

Preparing an SQL statement causes the command string to be parsed and converted into a set of byte-code commands. This byte-code is fed into SQLite's *Virtual Database Engine* (VDBE) for execution. The translation is not a consistent one-to-one affair. Depending on the database structure (such as indexes), the query optimizer may generate very different VDBE command sequences for similar SQL commands. The size and flexibility of the SQLite library can be largely attributed to the VDBE architecture.

To execute the VDBE code, the function `sqlite3_step()` is called. This function steps through the current VDBE command sequence until some type of program break is encountered. This can happen when a new row becomes available, or when the VDBE program reaches its end, indicating that no more data is available.

In the case of a `SELECT` query, `sqlite3_step()` will return once for each row in the result set. Each subsequent call to `sqlite3_step()` will continue execution of the statement until the next row is available or the statement reaches its end.

The function definition is quite simple:

`int sqlite3_step( sqlite3_stmt *stmt )`
: Attempts to execute the provided prepared statement. If a result set row becomes available, the function will return with a value of `SQLITE_ROW`. In that case, individual column values can be extracted with the `sqlite3_column_xxx()` functions. Additional rows can be returned by making further calls to `sqlite3_step()`. If the statement execution reaches its end, the code `SQLITE_DONE` will be returned. Once this happens, `sqlite3_step()` cannot be called again with this prepared statement until the statement is first reset using `sqlite3_reset()`.

If the first call to `sqlite3_step()` returns `SQLITE_DONE`, it means that the statement was successfully run, but there was no result data to make available. This is the typical case for most commands, other than `SELECT`. If `sqlite3_step()` is called repeatedly, a `SELECT` command will return `SQLITE_ROW` for each row of the result set before finally returning `SQLITE_DONE`. If a `SELECT` command returns no rows, it will return `SQLITE_DONE` on the first call to `sqlite3_step()`.

There are also some `PRAGMA` commands that will return a value. Even if the return value is a simple scalar value, that value will be returned as a one-row, one-column result set. This means that the first call to `sqlite3_step()` will return `SQLITE_ROW`, indicating result data is available. Additionally, if `PRAGMA count_changes` is set to true, the `INSERT`, `UPDATE`, and `DELETE` commands will return the number of rows they modified as a one-row, one-column integer value.

Any time `sqlite3_step()` returns `SQLITE_ROW`, new row data is available for processing. Row values can be inspected and extracted from the statement using the `sqlite3_column_xxx()` functions, which we will look at next. To resume execution of the statement, simply call `sqlite3_step()` again. It is common to call `sqlite3_step()` in a loop, processing each row until `SQLITE_DONE` is returned.

Rows are returned as soon as they are computed. In many cases, this spreads the processing costs out across all of the calls to `sqlite3_step()`, and allows the first row to be returned reasonably quickly. However, if the query has a `GROUP BY` or `ORDER BY` clause, the statement may be forced to first gather all of the rows within the result set before it is able to complete the final processing. In these cases, it may take a considerable time for the first row to become available, but subsequent rows should be returned very very quickly.

Result Columns

Any time `sqlite3_step()` returns the code `SQLITE_ROW`, a new result set row is available within the statement. You can use the `sqlite3_column_xxx()` functions to inspect and extract the column values from this row. Many of these functions require a column index parameter (`cidx`). Like C arrays, the first column in a result set always has an index of zero, starting from the left.

```
int sqlite3_column_count( sqlite3_stmt *stmt )
```

Returns the number of columns in the statement result. If the statement does not return values, a count of zero will be returned. Valid column indexes are zero through the count minus one. (`N` columns have the indexes `0` through `N-1`).

```
const char* sqlite3_column_name(   sqlite3_stmt *stmt, int cidx )
const void* sqlite3_column_name16( sqlite3_stmt *stmt, int cidx )
```

Returns the name of the specified column as a UTF-8 or UTF-16 encoded string. The returned string is the name provided by the `AS` clause within the `SELECT` header. For example, this function would return `person_id` for column zero of the SQL statement `SELECT pid AS person_id,...`. If no `AS` expression was given, the name is technically undefined and may change from one version of SQLite to another. This is especially true of columns that consist of an expression.

The returned pointers will remain valid until one of these functions is called again on the same column index, or until the statement is destroyed with `sqlite3_finalize()`. The pointers will remain valid (and unmodified) across calls to `sqlite3_step()` and `sqlite3_reset()`, as column names do not change from one execution to the next. These pointers should not be passed to `sqlite3_free()`.

```
int sqlite3_column_type( sqlite3_stmt *stmt, int cidx )
```

Returns the native type (storage class) of the value found in the specified column. Valid return codes can be `SQLITE_INTEGER`, `SQLITE_FLOAT`, `SQLITE_TEXT`, `SQLITE_BLOB`, or `SQLITE_NULL`. To get the correct native datatype, this function should be called before any attempt is made to extract the data.

This function returns the type of the actual value found in the current row. Because SQLite allows different types to be stored in the same column, the type returned for a specific column index may vary from row to row. This is also how you detect the presence of a NULL.

These `sqlite3_column_xxx()` functions allow your code to get an idea of what the available row looks like. Once you've figured out the correct value type, you can extract the value with one of these typed `sqlite3_column_xxx()` functions. All of these functions take the same parameters: a statement pointer and a column index.

`const void* sqlite3_column_blob( sqlite_stmt *stmt, int cidx )`
: Returns a pointer to the BLOB value from the given column. The pointer may be invalid if the BLOB has a length of zero bytes. The pointer may also be NULL if a type conversion was required.

`double sqlite3_column_double( sqlite_stmt *stmt, int cidx )`
: Returns a 64-bit floating-point value from the given column.

`int sqlite3_column_int( sqlite_stmt *stmt, int cidx )`
: Returns a 32-bit signed integer from the given column. The value will be truncated (without warning) if the column contains an integer value that cannot be represented in 32 bits.

`sqlite3_int64 sqlite3_column_int64( sqlite_stmt *stmt, int cidx )`
: Returns a 64-bit signed integer from the given column.

```
const unsigned char* sqlite3_column_text(   sqlite_stmt *stmt, int cidx )
const          void* sqlite3_column_text16( sqlite_stmt *stmt, int cidx )
```
: Returns a pointer to a UTF-8 or UTF-16 encoded string from the given column. The string will always be null-terminated, even if it is an empty string. Note that the returned `char` pointer is unsigned and will likely require a cast. The pointer may also be NULL if a type conversion was required.

`sqlite3_value* sqlite3_column_value( sqlite_stmt *stmt, int cidx )`
: Returns a pointer to an unprotected `sqlite3_value` structure. Unprotected `sqlite3_value` structures cannot safely undergo type conversion, so you should not attempt to extract a primitive value from this structure using the `sqlite3_value_xxx()` functions. If you want a primitive value, you should use one of the other `sqlite3_column_xxx()` functions. The only safe use for the returned pointer is to call `sqlite3_bind_value()` or `sqlite3_result_value()`. The first is used to bind the value to another prepared statement, while the second is used to return a value in a user-defined SQL function (see "Binding Values" on page 135, or "Returning Results and Errors" on page 186).

There is no `sqlite3_column_null()` function. There is no need for one. If the native datatype is NULL, there is no additional value or state information to extract.

Any pointers returned by these functions become invalid if another call to any `sqlite3_column_xxx()` function is made using the same column index, or when `sqlite3_step()` is next called. Pointers will also become invalid if the statement is reset or finalized. SQLite will take care of all the memory management associated with these pointers.

If you request a datatype that is different from the native value, SQLite will attempt to convert the value. Table 7-1 describes the conversion rules used by SQLite.

Table 7-1. SQLite type conversion rules.

Original type	Requested type	Converted value
NULL	Integer	0
NULL	Float	0.0
NULL	Text	NULL pointer
NULL	BLOB	NULL pointer
Integer	Float	Converted float
Integer	Text	ASCII number
Integer	BLOB	Same as text
Float	Integer	Rounds towards zero
Float	Text	ASCII number
Float	BLOB	Same as text
Text	Integer	Internal `atoi()`
Text	Float	Internal `atof()`
Text	BLOB	No change
BLOB	Integer	Converts to text, `atoi()`
BLOB	Float	Converts to text, `atof()`
BLOB	Text	Adds terminator

Some conversions are done in place, which can cause subsequent calls to `sqlite3_column_type()` to return undefined results. That's why it is important to call `sqlite3_column_type()` before trying to extract a value, unless you already know exactly what datatype you want.

Although numeric values are returned directly, text and BLOB values are returned in a buffer. To determine how large that buffer is, you need to ask for the byte count. That can be done with one of these two functions.

`int sqlite3_column_bytes( sqlite3_stmt *stmt, int cidx )`
: Returns the number of *bytes* in a BLOB or in a UTF-8 encoded text value. If returning the size of a text value, the size will include the terminator.

`int sqlite3_column_bytes16( sqlite3_stmt *stmt, int cidx )`
: Returns the number of *bytes* in a UTF-16 encoded text value, including the terminator.

Be aware that these functions can cause a data conversion in text values. That conversion can invalidate any previously returned pointer. For example, if you call `sqlite3_column_text()` to get a pointer to a UTF-8 encoded string, and then call `sqlite3_column_bytes16()` on the same column, the internal column value will be converted from a UTF-8 encoded string to a UTF-16 encoded string. This will invalidate the character pointer that was originally returned by `sqlite3_column_text()`.

Similarly, if you first call `sqlite3_column_bytes16()` to get the size of UTF-16 encoded string, and then call `sqlite3_column_text()`, the internal value will be converted to a UTF-8 string before a string pointer is returned. That will invalidate the length value that was originally returned.

The easiest way to avoid problems is to extract the datatype you want and then call the matching bytes function to find out how large the buffer is. Here are examples of safe call sequences:

```
/* correctly extract a blob */
buf_ptr = sqlite3_column_blob( stmt, n );
buf_len = sqlite3_column_bytes( stmt, n );

/* correctly extract a UTF-8 encoded string */
buf_ptr = sqlite3_column_text( stmt, n );
buf_len = sqlite3_column_bytes( stmt, n );

/* correctly extract a UTF-16 encoded string */
buf_ptr = sqlite3_column_text16( stmt, n );
buf_len = sqlite3_column_bytes16( stmt, n );
```

By matching the correct bytes function for your desired datatype, you can avoid any type conversions keeping both the pointer and length valid and correct.

You should always use `sqlite3_column_bytes()` to determine the size of a BLOB.

Reset and Finalize

When a call to `sqlite3_step()` returns `SQLITE_DONE`, the statement has successfully finished execution. At that point, there is nothing further you can do with the statement. If you want to use the statement again, it must first be reset.

`int sqlite3_reset( sqlite3_stmt *stmt )`
: Resets a prepared statement so that it is ready for another execution. A statement should be reset as soon as you're done using it. This will ensure any locks are released.

The function `sqlite3_reset()` can be called any time after `sqlite3_step()` is called. It is valid to call `sqlite3_reset()` before a statement is finished executing (that is, before `sqlite3_step()` returns `SQLITE_DONE` or an error indicator). You can't cancel a running `sqlite3_step()` call this way, but you can short-circuit the return of additional `SQLITE_ROW` values.

For example, if you only want the first six rows of a result set, it is perfectly valid to call `sqlite3_step()` only six times and then reset the statement, even if `sqlite3_step()` would continue to return `SQLITE_ROW`.

The function `sqlite3_reset()` simply resets a statement, it does not release it. To destroy a prepared statement and release its memory, the statement must be finalized.

`int sqlite3_finalize( sqlite3_stmt *stmt )`
: Destroys a prepared statement and releases any associated resources.

The function `sqlite3_finalize()` can be called at any time on any statement that was successfully prepared. All of the prepared statements associated with a database connection must be finalized before the database connection can be closed.

Although both of these functions can return errors, they always perform their function. Any error that is returned was generated by the last call to `sqlite3_step()`. See "Result Codes and Error Codes" on page 146 for more details.

It is a good idea to reset or finalize a statement as soon as you are done using it. A call to `sqlite3_reset()` or `sqlite3_finalize()` ensures the statement will release any locks it might be holding, and frees any resources associated with the prior statement execution. If an application keeps statements around for an extended period of time, they should be kept in a reset state, ready to be bound and executed.

Statement Transitions

Prepared statements have a significant amount of state. In addition to the currently bound parameter values and other details, every prepared statement is always in one of three major states. The first is the "ready" state. Any freshly prepared or reset statement will be "ready." This indicates that the statement is ready to execute, but hasn't been started. The second state is "running," indicating that a statement has started to execute, but hasn't yet finished. The final state is "done," which indicates the statement has completed executing.

Knowing the current state of a statement is important. Although some API functions can be called at any time (like `sqlite3_reset()`), other API functions can only be called when a statement is in a specific state. For example, the `sqlite3_bind_xxx()` functions can only be called when a statement is in its "ready" state. Figure 7-1 shows the different states and how a statement transitions from one state to another.

There is no way to query the current state of a statement. Transitions between states are normally controlled by the design and flow of the application.

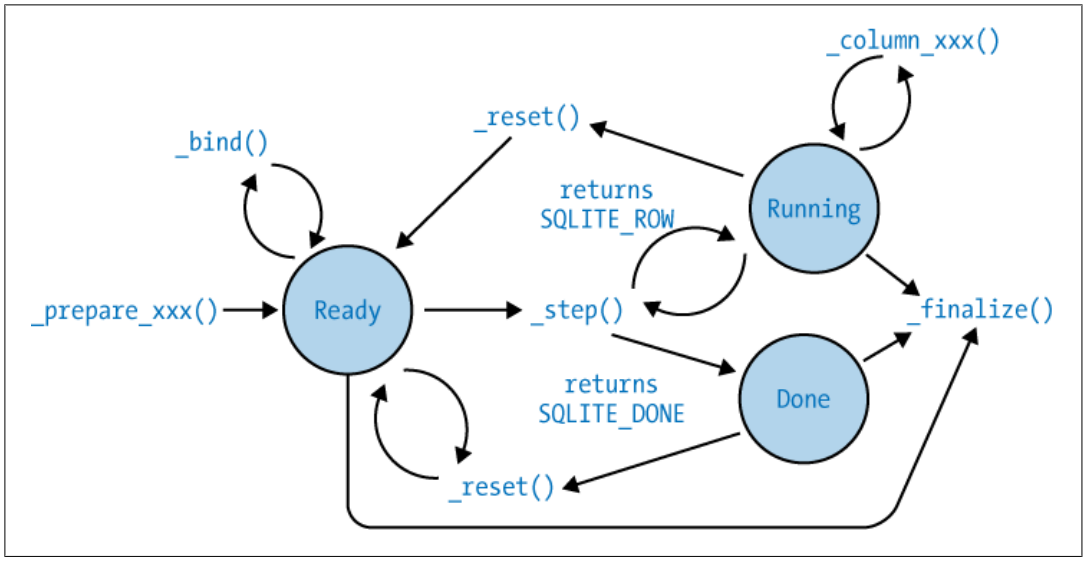


Figure 7-1. Prepared statement transitions. A statement can be in one of three states. Depending on the current state, only some API functions are valid. Calling a function in an inappropriate state will result in an SQLITE_MISUSE error.

Examples

Here are two examples of using prepared statements. The first example executes a `CREATE TABLE` statement by first preparing the SQL string and then calling `sqlite3_step()` to execute the statement:

```
sqlite3_stmt    *stmt = NULL;

/* ... open database ... */

rc = sqlite3_prepare_v2( db, "CREATE TABLE tbl ( str TEXT )", -1, &stmt, NULL );
if ( rc != SQLITE_OK) exit( -1 );

rc = sqlite3_step( stmt );
if ( rc != SQLITE_DONE ) exit ( -1 );

sqlite3_finalize( stmt );

/* ... close database ... */
```

The `CREATE TABLE` statement is a DDL command that does not return any type of value and only needs to be "stepped" once to fully execute the command. Remember to reset or finalize statements as soon as they're finished executing. Also remember that all statements associated with a database connection must be fully finalized before the connection can be closed.

This second example is a bit more complex. This code performs a `SELECT` and loops over `sqlite3_step()` extracting all of the rows in the table. Each value is displayed as it is extracted:

```
const char      *data = NULL;
sqlite3_stmt    *stmt = NULL;

/* ... open database ... */

rc = sqlite3_prepare_v2( db, "SELECT str FROM tbl ORDER BY 1", -1, &stmt, NULL );
if ( rc != SQLITE_OK) exit( -1 );

while( sqlite3_step( stmt ) == SQLITE_ROW ) {
    data = (const char*)sqlite3_column_text( stmt, 0 );
    printf( "%s\n", data ? data : "[NULL]" );
}

sqlite3_finalize( stmt );

/* ... close database ... */
```

This example does not check the type of the column value. Since the value will be displayed as a string, the code depends on SQLite's internal conversion process and always requests a text value. The only tricky bit is that the string pointer may be NULL, so we need to be prepared to deal with that in the `printf()` statement.

Bound Parameters

Statement parameters are special tokens that are inserted into the SQL command string before it is passed to one of the `sqlite3_prepare_xxx()` functions. They act as a placeholder for any literal value, such as a bare number or a single quote string. After the statement is prepared, but before it is executed, you can bind specific values to each statement parameter. Once you're done executing a statement, you can reset the statement, bind new values to the parameters, and execute the statement again—only this time with the new values.

Parameter Tokens

SQLite supports five different styles of statement parameters. These short string tokens are placed directly into the SQL command string, which can then be passed to one of the `sqlite3_prepare_xxx()` functions. Once the statement is prepared, the individual parameters are referenced by index.

`?`

: An anonymous parameter with automatic index. As the statement is processed, each anonymous parameter is assigned a unique, sequential index value, starting with one.

`?<index>`
: Parameter with explicit numeric index. Duplicate indexes allow the same value to be bound multiple places in the same statement.

`:<name>`
: A named parameter with an automatic index. Duplicate names allow the same value to be bound multiple places in the same statement.

`@<name>`
: A named parameter with an automatic index. Duplicate names allow the same value to be bound multiple places in the same statement. Works exactly like the colon parameter.

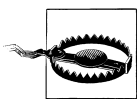
`$<name>`
: A named parameter with an automatic index. Duplicate names allow the same value to be bound multiple places in the same statement. This is an extended syntax to support Tcl variables. Unless you're doing Tcl programming, I suggest you use the colon format.

To get an idea of how these work, consider this `INSERT` statement:

```
INSERT INTO people (id, name) VALUES ( ?, ? );
```

The two statement parameters represent the `id` and `name` values being inserted. Parameter indexes start at one, so the first parameter that represents the `id` value has an index of one, and the parameter used to reference the `name` value has an index of two.

Notice that the second parameter, which is likely a text value, does not have single quotes around it. The single quotes are part of the string-literal representation, and are not required for a parameter value.



Statement parameters should not be put in quotes. The notation `'?'` designates a one-character text value, not a parameter.

Once this statement has been prepared, it can be used to insert multiple rows. For each row, simply bind the appropriate values, step through the statement, and then reset the statement. After the statement has been reset, new values can be bound to the parameters and the statement can be stepped again.

You can also use explicit index values:

```
INSERT INTO people (id, name) VALUES ( ?1, ?2 );
```

Using explicit parameter indexes has two major advantages. First, you can have multiple instances of the same index value, allowing the same value to be bound to more than one place in the same statement.

Second, explicit indexes allow the parameters to appear out of order. There can even be gaps in the index sequence. This can help simplify application code maintenance if the query is modified and parameters are added or removed.

This level of abstraction can be taken even further by using named parameters. In this case, you allow SQLite to assign parameter index values as it sees fit, in a similar fashion to anonymous parameters. The difference is that you can ask SQLite to tell you the index value of a specific parameter based off the name you've given it. Consider this statement:

```
INSERT INTO people (id, name) VALUES ( :id, :name );
```

In this case, the parameter values are quite explicit. As we will see in the next section, the code that binds values to these parameters is also quite explicit, making it very clear what is going on. Best of all, it doesn't matter if new parameters are added. As long as the existing names remain unchanged, the code will properly find and bind the named parameters.

Note, however, that parameters can only be used to replace literal values, such as quoted strings or numeric values. Parameters cannot be used in place of identifiers, such as table names or column names. The following bit of SQL is invalid:

```
SELECT * FROM ?;   -- INCORRECT: Cannot use a parameter as an identifier
```

If you attempt to prepare this statement, it will fail. This is because the parameter (which acts as an unknown literal value) is being used where an identifier is required. This is invalid, and the statement will not prepare correctly.

Within a statement, it is best to choose a specific parameter style and stick with it. Mixing anonymous parameters with explicit indexes or named parameters is likely to cause confusion about what index belongs to which parameter.

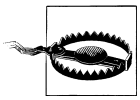
Personally, I prefer to used the colon-name-style parameters. Using named parameters eliminates the need to know any specific index values, allowing you to just reference the name at runtime. The use of short, significant names can also make the intent of both your SQL statements and your bind code easier to understand.

Binding Values

When you first prepare a statement with parameters, all of the parameters start out with a NULL assigned to them. Before you execute the statement, you can bind specific values to each parameter using the `sqlite3_bind_xxx()` family of functions.

There are nine `sqlite3_bind_xxx()` functions available, plus a number of utility functions. These functions can be called any time after the statement is prepared, but before `sqlite3_step()` is called for the first time. Once `sqlite3_step()` has been called, these functions cannot be called again until the statement is reset.

All the `sqlite3_bind_xxx()` functions have the same first and second parameters and return the same result. The first parameter is always a pointer to an `sqlite3_stmt`, and the second is the index of the parameter to bind. Remember that for anonymous parameters, the first index value starts with one. For the most part, the third parameter is the value to bind. The fourth parameter, if present, indicates the length of the data value in bytes. The fifth parameter, if present, is a function pointer to a memory management callback.



Remember that bind index values start with one (1), unlike result column indexes, which start with zero (0).

All the bind functions return an integer error code, which is equal to `SQLITE_OK` upon success.

The bind functions are:

```
int sqlite3_bind_blob( sqlite3_stmt *stmt, int pidx,
                       const void *data, int data_len, mem_callback )
```
Binds an arbitrary length binary data BLOB.

```
int sqlite3_bind_double( sqlite3_stmt *stmt, int pidx, double data )
```
Binds a 64-bit floating point value.

```
int sqlite3_bind_int(   sqlite3_stmt *stmt, int pidx, int data )
```
Binds a 32-bit signed integer value.

```
int sqlite3_bind_int64( sqlite3_stmt *stmt, int pidx, sqlite3_int64 )
```
Binds a 64-bit signed integer value.

```
int sqlite3_bind_null( sqlite3_stmt *stmt, int pidx )
```
Binds a NULL datatype.

```
int sqlite3_bind_text( sqlite3_stmt *stmt, int pidx,
                       const char *data, int data_len, mem_callback )
```
Binds an arbitrary length UTF-8 encoded text value. The length is in bytes, not characters. If the length parameter is negative, SQLite will compute the length of the string up to, but not including, the null terminator. It is recommended that the manually computed lengths do not include the terminator (the terminator will be included when the value is returned).

```
int sqlite3_bind_text16( sqlite3_stmt *stmt, int pidx,
                         const void *data, int data_len, mem_callback )
```
Binds an arbitrary length UTF-16 encoded text value. The length is in bytes, not characters. If the length parameter is negative, SQLite will compute the length of the string up to, but not including, the null terminator. It is recommended that the manually computed lengths do not include the terminator (the terminator will be included when the value is returned).

`int sqlite3_bind_zeroblob( sqlite3_stmt *stmt, int pidx, int len )`
: Binds an arbitrary length binary data BLOB, where each byte is set to zero (`0x00`). The only additional parameter is a length value, in bytes. This function is particularly useful for creating large BLOBs that can then be updated with the incremental BLOB interface. See `sqlite3_blob_open()` in Appendix G for more details.

In addition to these type-specific bind functions, there is also a specialized function:

```
int sqlite3_bind_value( sqlite3_stmt *stmt, int pidx,
                        const sqlite3_value *data_value )
```
: Binds the type and value of an `sqlite3_value` structure. An `sqlite3_value` structure can hold any data format.

The text and BLOB variants of `sqlite3_bind_xxx()` require you to pass a buffer pointer for the data value. Normally this buffer and its contents must remain valid until a new value is bound to that parameter, or the statement is finalized. Since that might be some time later in the code, these bind functions have a fifth parameter that controls how the buffer memory is handled and possibly released.

If the fifth parameter is either NULL or the constant `SQLITE_STATIC`, SQLite will take a hands-off approach and assume the buffer memory is either static or that your application code is taking care of maintaining and releasing any memory.

If the fifth parameter is the constant `SQLITE_TRANSIENT`, SQLite will make an internal copy of the buffer. This allows you to release your buffer immediately (or allow it to go out of scope, if it happens to be on the stack). SQLite will automatically release the internal buffer at an appropriate time.

The final option is to pass a valid `void mem_callback( void* ptr )` function pointer. This callback will be called when SQLite is done with the buffer and wants to release it. If the buffer was allocated with `sqlite3_malloc()` or `sqlite3_realloc()`, you can pass a reference to `sqlite3_free()` directly. If you allocated the buffer with a different set of memory management calls, you'll need to pass a reference to a wrapper function that calls the appropriate memory release function.

Once a value has been bound to a parameter, there is no way to extract that value back out of the statement. If you need to reference a value after it has been bound, you must keep track of it yourself.

To help you figure out what parameter index to use, there are three utility functions:

`int sqlite3_bind_parameter_count( sqlite3_stmt *stmt )`
: Returns an integer indicating the largest parameter index. If no explicit numeric indexes are used (`?<number>`), this will the be the number of unique parameters that appear in a statement. If explicit numeric indexes are used, there may be gaps in the number sequence.

`int sqlite3_bind_parameter_index( sqlite3_stmt *stmt, const char *name )`
: Returns the index of a named parameter. The name must include any leading character (such as ":") and must be given in UTF-8, even if the statement was prepared from UTF-16. A zero is returned if a parameter with a matching name cannot be found.

`const char* sqlite3_bind_parameter_name( sqlite3_stmt *stmt, int pidx )`
: Returns the full text representation of a specific parameter. The text is always UTF-8 encoded and includes the leading character.

Using the `sqlite3_bind_parameter_index()` function, you can easily find and bind named parameters. The `sqlite3_bind_xxx()` functions will properly detect an invalid index range, allowing you to look up the index and bind a value in one line:

```
sqlite3_bind_int(stmt, sqlite3_bind_parameter_index(stmt, ":pid"), pid);
```

If you want to clear all of the bindings back to their initial NULL defaults, you can use the function `sqlite3_clear_bindings()`:

`int sqlite3_clear_bindings( sqlite3_stmt *stmt )`
: Clears all parameter bindings in a statement. After calling, all parameters will have a NULL bound to them. This will cause the memory management callback to be called on any text or BLOB values that were bound with a valid function pointer. Currently, this function always returns `SQLITE_OK`.

If you want to be absolutely sure bound values won't leak from one statement execution to the next, it is best to clear the bindings any time you reset the statement. If you're doing manual memory management on data buffers, you can free any memory used by bound values after this function is called.

Security and Performance

There are significant security advantages to using bound parameters. Many times people will manipulate SQL strings to substitute the values they want to use. For example, consider building an SQL statement in C using the string function `snprintf()`:

```
snprintf(buf, buf_size,
         "INSERT INTO people( id, name ) VALUES ( %d, '%s' );",
         id_val, name_val);
```

In this case we do need single quotes around the string value, as we're trying to form a literal representation. If we pass in these C values:

```
id_val = 23;
name_val = "Fred";
```

Then we get the following SQL statement in our buffer:

```
INSERT INTO people( id, name ) VALUES ( 23, 'Fred');
```

This seems simple enough, but the danger with a statement like this is that the variables need to be sanitized before they're passed into the SQL statement. For example, consider these values:

```
id_val = 23;
name_val = "Fred' ); DROP TABLE people;";
```

This would cause our `snprintf()` to create the following SQL command sequence, with the individual commands split out onto their own lines for clarity:

```
INSERT INTO people( id, name ) VALUES ( 23, 'Fred' );
DROP TABLE people;
' );
```

While that last statement is nonsense, the second statement is cause for concern.

Thankfully, things are not quite as bad as they seem. The `sqlite3_prepare_xxx()` functions will only prepare a single statement (up to the first semicolon), unless you explicitly pass the remainder of the SQL command string to another `sqlite3_prepare_xxx()` call. That limits what can be done in a case like this, unless your code automatically prepares and executes multiple statements from a single command buffer.

Be warned, however, that the interfaces provided by many scripting languages will do exactly that, and will automatically process multiple SQL statements passed in with a single call. The SQLite convenience functions, including `sqlite3_exec()`, will also automatically process multiple SQL commands passed in through a single string. What makes `sqlite3_exec()` particularly dangerous is that the convenience functions don't allow the use of bound values, forcing you to programmatically build SQL command statements and opening you up to problems. Later in the chapter, we'll take a closer look at `sqlite3_exec()` and why it usually isn't the best choice.

Even if SQLite will only process the first command, damage can still be done with subqueries and other commands. Bad input can also force a statement to fail. Consider the result if the name value is:

```
Fred', 'extra junk
```

If you're updating a series of records based off this `id` value, you had better wrap all the commands up in a transaction and be prepared to roll it back if you encounter an error. If you just assume the commands will work, you'll end up with an inconsistent database.

This type of attack is known as an *SQL injection attack*. An SQL injection attack inserts SQL command fragments into data values, causing the database to execute arbitrary SQL commands. Unfortunately, it is *extremely* common for websites to be susceptible to this kind of attack. It also borders on inexcusable, because it is typically very easy to avoid.

One defense against SQL injections is to try to sanitize any string values received from an untrusted source. For example, you might try to substitute all single quote characters with two single quote characters (the standard SQL escape mechanism). This can get quite complex, however, and you're putting utter faith in the code's ability to correctly sanitize untrusted strings.

A much easier way to defend yourself against SQL injections is to use SQL statement parameters. Injection attacks depend on a data value being represented as a literal value in an SQL command statement. The attack only works if the attack value is passed through the SQL parser, where it alters the meaning of the surrounding SQL commands.

In the case of SQL parameters, the bound values are never passed through the SQL parser. An SQL statement is only parsed when the command is prepared. If you're using parameters, the SQL engine parses only the parameter tokens. Later, when you bind a value to a specific parameter, that value is bound directly in its native format (i.e. string, integer, etc.) and is not passed through the SQL parser. As long as you're careful about how you extract and display the string, it is perfectly safe to directly bind an untrusted string value to a parameter value without fear of an SQL injection.

Besides avoiding injection attacks, parameters can also be faster and use less memory than string manipulations. Using a function such as `snprintf()` requires an SQL command template, and a sufficiently large output buffer. The string manipulation functions also need working memory, plus you may need additional buffers to copy and sanitize values. Additionally, a number of datatypes, such as integers and floating-point numbers (and especially BLOBs), often take up significantly more space in their string representation. This further increases memory usage. Finally, once the final command buffer has been created, all the data needs to be passed through the SQL parser, where the literal data values are converted back into their native format and stored in additional buffers.

Compare that to the resource usage of preparing and binding a statement. When using parameters, the SQL command statement is essentially static, and can be used as is, without modification or additional buffers. The parser doesn't need to deal with converting and storing literal values. In fact, the data values normally never leave their native format, further saving time and memory by avoiding conversion in and out of a string representation.

Using parameters is the safe and wise choice, even for situations when a statement is only used once. It may take a few extra lines of code, but the process will be safer, faster, and more memory efficient.

Example

This example executes an `INSERT` statement. Although this statement is only executed once, it still uses bind parameters to protect against possible injection attacks. This eliminates the need to sanitize the input value.

The statement is first prepared with a statement parameter. The data value is then bound to the statement parameter before we execute the prepared statement:

```
char            *data = ""; /* default to empty string */
sqlite3_stmt    *stmt = NULL;
int             idx = -1;

/* ... set "data" pointer ... */
/* ... open database ... */

rc = sqlite3_prepare_v2( db, "INSERT INTO tbl VALUES ( :str )", -1, &stmt, NULL );
if ( rc != SQLITE_OK) exit( -1 );

idx = sqlite3_bind_parameter_index( stmt, ":str" );
sqlite3_bind_text( stmt, idx, data, -1, SQLITE_STATIC );

rc = sqlite3_step( stmt );
if (( rc != SQLITE_DONE )&&( rc != SQLITE_ROW )) exit ( -1 );

sqlite3_finalize( stmt );

/* ... close database ... */
```

In this case we look for either an `SQLITE_DONE` or an `SQLITE_ROW` return value. Both are possible. Although the `INSERT` itself will be fully executed on the first call to `sqlite3_step()`, if `PRAGMA count_changes` is enabled, then the statement may return a value. In this case, we want to ignore any potential return value without triggering an error, so we must check for both possible return codes. For more details, see `count_changes` in Appendix F.

Potential Pitfalls

It is important to understand that all parameters must have some literal associated with them. As soon as you prepare a statement, all the parameters are set to NULL. If you fail to bind an alternate value, the parameter still has a literal NULL associated with it. This has a few ramifications that are not always obvious.

The general rule of thumb is that bound parameters act as literal string substitutions. Although they offer additional features and protections, if you're trying to figure out the expected behavior of a parameter substitution, it is safe to assume you'll get the exact same behavior as if the parameter was a literal string substitution.

In the case of an `INSERT` statement, there is no way to force a default value to be used. For example, if you have the following statement:

```
INSERT INTO membership ( pid, gid, type ) VALUES ( :pid, :gid, :type );
```

Even if the `type` column has a default value available, there is no way this statement can use it. If you fail to bind a value to the `:type` parameter, a NULL will be inserted, rather than the default value. The only way to insert a default value into the type column is to use a statement that doesn't reference it, such as:

```
INSERT INTO membership ( pid, gid ) VALUES ( :pid, :gid );
```

This means that if you're heavily dependent on database-defined default values, you may need to prepare several variations of an `INSERT` statement to cover the different cases when different data values are available. Of course, if your application code is aware of the proper default values, it can simply bind that value to the proper parameter.

The other area where parameters can cause surprises is in NULL comparisons. For example, consider the statement:

```
SELECT * FROM employee WHERE manager = :manager;
```

This works for normal values, but if a NULL is bound to the `:manager` parameter, no rows will ever be returned. If you need the ability to test for a NULL in the `manager` column, make sure you use the `IS` operator:

```
SELECT * FROM employee WHERE manager IS :manager;
```

For more details, see `IS` in Appendix D.

This behavior also makes it tricky to "stack" conditionals. For example, if you have the statement:

```
SELECT * FROM employee WHERE manager = :manager AND project = :project;
```

you must provide a meaningful value for both `:manager` and `:project`. If you want the ability to search on a `manager`, on a `project`, or on a `manager` and a `project`, you need to prepare multiple statements, or you need to add a bit more logic:

```
...WHERE ( manager = :manager OR :manager IS NULL )
     AND ( project = :project OR :project IS NULL );
```

This query will ignore one (or both) of the value conditions if you assign a NULL to the appropriate parameters. This expression won't let you explicitly search for NULLs, but that can be done with additional parameters and logic. Preparing more flexible statements reduces the number of unique statements you need to manage, but it also tends to make them more complex and can make them run slower. If they get too complex, it might make more sense to simply define a new set of statements, rather than adding more and more parameter logic to the same statement.

Convenience Functions

SQLite includes a number of convenience functions that can be used to prepare, step, and finalize an SQL statement in one call. Most of these functions exist for historical reasons and, as the name says, convenience.

While they're not fully deprecated, there are a number of reasons why their use is not exactly encouraged. First off, understand that there is nothing special under the hood. Both of these functions eventually call the same `sqlite3_prepare_xxx()`, `sqlite3_step()`, and `sqlite3_finalize()` calls that are available in the public API. These functions are not faster, nor are they more efficient.

Second, since the API doesn't support the use of bound parameters, you're forced to use string manipulations to build your SQL commands. That means these functions are slower to process and much more vulnerable to SQL injection attacks. This is particularly dangerous because all the convenience functions are designed to automatically process multiple SQL statements from a single command string. If input strings are not properly sanitized, this situation effectively gives anyone providing input data full access to the database engine, including the ability to delete data or drop whole tables.

These functions also tend to be a bit slower. All results are returned in a string representation, without any kind of type information. This can make it difficult to determine the type of a return value, and can lead to a lot of extra type conversions.

For all their disadvantages, there is still the simple fact that these functions are very convenient. If you're just trying to throw together a quick and dirty snippet of code, these functions provide an easy means of doing that. They're also perfectly acceptable for DDL commands, such as `CREATE TABLE`. For any type of DML command, especially those that involve values from unsanitized sources, I strongly recommend using the normal prepare, step, and finalize routines. You'll end up with safer code and better performance.

The first function allows for fairly generic execution of any SQL command string.

```
int sqlite3_exec( sqlite3 *db, const char *sql,
                  callback_ptr, void *userData, char **errMsg )
```

Prepares and executes one or more SQL statements, calling the optional callback for each result set row for each statement. The first parameter is a valid database connection. The second parameter is a UTF-8 encoded string that consists of one or more SQL statements. The third parameter is a pointer to a callback function. The prototype of this function is given below. This function pointer can be NULL. The fourth parameter is a user-data pointer that will be passed to the callback. The value can be whatever you want, including NULL. The fifth parameter is a reference to a character pointer. If an error is generated and this parameter is non-NULL, `sqlite3_exec()` will allocate a string buffer and return it. If the passed-back pointer is non-NULL, you are responsible for releasing the buffer with `sqlite3_free()` once you are done with it.

If the SQL string consists of multiple SQL statements separated by semicolons, each statement will be executed in turn.

If the call is successful and all statements are processed without errors, `SQLITE_OK` will be returned. Otherwise, just about any of the other return codes are possible, since this one function runs through the whole statement preparation and execution process.

The `sqlite3_exec()` function is reasonably all encompassing, and can be used to execute any SQL statement. If you're executing a table query and want to access the result set, you will need to supply a function pointer that references a user-defined callback. This callback will be called once for each row returned. If you're executing an SQL statement that does not normally return any database value, there is no need to provide a callback function. The success or failure of the SQL command will be indicated in the return value.

The `sqlite3_exec()` function makes any database results available through a user-defined callback function. As each result row is computed, the callback is called to make the row data available to your code. Essentially, each internal call to `sqlite3_step()` that results in a return value of `SQLITE_ROW` results in a callback.

The format of the callback looks like this:

```
int user_defined_exec_callback( void *userData, int numCol,
                                char **colData, char **colName )
```

This function is not part of the SQLite API. Rather, this shows the required format for a user-defined `sqlite3_exec()` callback. The first parameter is the user-data pointer passed in as the fourth parameter to `sqlite3_exec()`. The second parameter indicates how many columns exist in this row. The third and fourth parameters both return an array of strings (`char` pointers). The third parameter holds the data values for this row, while the forth parameter holds the column names. All values are returned as strings. There is no type information.

Normally, the callback should return a zero value. If a nonzero value is returned, execution is stopped and `sqlite3_exec()` will return `SQLITE_ABORT`.

The second, third, and fourth parameters act very similar to the traditional C variables `argc` and `argv` (and an extra `argv`) in `main( int argc, char **argv )`, the traditional start to every C program. The column value and name arrays will always be the same size for any given callback, but the specific size of the arrays and the column names can change over the course of processing a multi-statement SQL string. There is no need to release any of these values. Once your callback function returns, `sqlite3_exec()` will handle all the memory management.

If you'd prefer not to mess with a callback, you can use `sqlite3_get_table()` to extract a whole table at once. Be warned, however, that this can consume a large amount of memory, and must be used carefully.

While you can technically call `sqlite3_get_table()` with any SQL command string, it is specifically designed to work with `SELECT` statements.

```
int sqlite3_get_table( sqlite3 *db, const char *sql, char ***result,
                       int *numRow, int *numCol, char **errMsg );
```

Prepares and executes an SQL command string, consisting of one or more SQL statements. The full contents of the result set(s) is returned in an array of UTF-8 strings.

The first parameter is a database connection. The second parameter is a UTF-8 encoded SQL command string that consists of one or more SQL statements. The third parameter is a reference to a one-dimensional array of strings (`char` pointers). The results of the query are passed back through this reference. The fourth and fifth parameters are integer references that pass back the number of rows and the number of columns, respectively, in the result array. The sixth and final parameter is a reference to a character string, and is used to return any error message.

The result array consists of `(numCol * (numRow + 1))` entries. Entries zero through *`numCol - 1`* hold the column names. Each additional set of `numCol` entries holds one row worth of data.

If the call is successful and all statements are processed without errors, `SQLITE_OK` will be returned. Otherwise, just about any of the other return codes are possible, since this one function runs through the whole statement preparation and execution process.

```
void sqlite3_free_table( char **result )
```

Correctly frees the memory allocated by a successful call to `sqlite3_get_table()`. Do not attempt to free this memory yourself.

As indicated, you must release the result of a call to `sqlite3_get_table()` with a call to `sqlite3_free_table()`. This will properly release the individual allocations used to build the result value. As with `sqlite3_exec()`, you must call `sqlite3_free()` on any `errMsg` value that is returned.

The result array is a one-dimensional array of character pointers. You must compute your own offsets into the array using the formula:

```
/* offset to access column C of row R of **result */
int  offset  = ((R + 1) * numCol) + C;
char *value  = result[offset];
```

The "`+ 1`" used to compute the row offset is required to skip over the column names, which are stored in the first row of the result. This assumes that the first row and column would be accessed with an index of zero.

As a convenience function, there is nothing special about `sqlite3_get_table()`. In fact, it is just a wrapper around `sqlite3_exec()`. It offers no additional performance benefits over the prepare, step, and finalize interfaces. In fact, between all the type conversions inherent in `sqlite3_exec()`, and all the memory allocations, `sqlite3_get_table()` has substantial overhead over other methods.

Since `sqlite3_get_table()` is a wrapper around `sqlite3_exec()`, it is possible to pass in an SQL command string that consists of multiple SQL statements. In the case of `sqlite3_get_table()`, this must be done with care, however.

If more than one `SELECT` statement is passed in, there is no way to determine where one result set ends and the next begins. All the resulting rows are run together as one large result array. All of the statements must return the same number of columns, or the whole `sqlite3_get_table()` command will fail. Additionally, only the first statement will return any column names. To avoid these issues, it is best to call `sqlite3_get_table()` with single SQL commands.

There are a number of reasons why these convenience functions may not be the best choice. Their use requires building an SQL command statement using string manipulation functions, and that process tends to be error prone. However, if you insist, your best bet is to use one of SQLite's built-in string-building functions: `sqlite3_mprintf()`, `sqlite3_vmprintf()`, or `sqlite3_snprintf()`. See Appendix G for more details.

Result Codes and Error Codes

You may have noticed that I've been fairly quiet about the result codes that can be expected from a number of these API calls. Unfortunately, error handling in SQLite is a bit complex. At some point, it was recognized that the original error reporting mechanism was a bit too generic and somewhat difficult to use. To address these concerns, a newer "extended" set of error codes was added, but this new system had to be layered on top of the existing system without breaking backward compatibility. As a result, we have both the older and newer error codes, as well as specific API calls that will alter the meaning of some of the codes. This all makes for a somewhat complex situation.

Standard Codes

Before we get into when things go wrong, let's take a quick look at when things go right. Generally, any API call that simply needs to indicate, "that worked," will return the constant `SQLITE_OK`. Not all non-`SQLITE_OK` return codes are errors, however. Recall that `sqlite3_step()` returns `SQLITE_ROW` or `SQLITE_DONE` to indicate specific return state.

Table 7-2 provides a quick overview of the standard error codes. At this point in the development life cycle, it is unlikely that additional standard error codes will be added. Additional extended error codes may be added at any time, however.

Table 7-2. SQLite standard return codes

Return code constant	Return code meaning
`SQLITE_OK`	Operation successful
`SQLITE_ERROR`	Generic error
`SQLITE_INTERNAL`	Internal SQLite library error

Return code constant	Return code meaning
SQLITE_PERM	Access permission denied
SQLITE_ABORT	User code or SQL requested an abort
SQLITE_BUSY	A database file is locked (usually recoverable)
SQLITE_LOCKED	A table is locked
SQLITE_NOMEM	Memory allocation failed
SQLITE_READONLY	Attempted to write to a read-only database
SQLITE_INTERRUPT	sqlite3_interrupt() was called
SQLITE_IOERR	Some type of I/O error
SQLITE_CORRUPT	Database file is malformed
SQLITE_FULL	Database is full
SQLITE_CANTOPEN	Unable to open requested database file
SQLITE_EMPTY	Database file is empty
SQLITE_SCHEMA	Database schema has changed
SQLITE_TOOBIG	TEXT or BLOB exceeds limit
SQLITE_CONSTRAINT	Abort due to constraint violation
SQLITE_MISMATCH	Datatype mismatch
SQLITE_MISUSE	API used incorrectly
SQLITE_NOLFS	Host OS cannot provide required functionality
SQLITE_AUTH	Authorization denied
SQLITE_FORMAT	Auxiliary database format error
SQLITE_RANGE	Bad bind parameter index
SQLITE_NOTADB	File is not a database

Many of these errors are fairly specific. For example, SQLITE_RANGE will only be returned by one of the sqlite3_bind_xxx() functions. Other codes, like SQLITE_ERROR, provide almost no information about what went wrong.

Of specific interest to the developer is SQLITE_MISUSE. This indicates an attempt to use a data structure or API call in an incorrect or otherwise invalid way. For example, trying to bind a new value to a prepared statement that is in the middle of an sqlite3_step() sequence would result in a misuse error. Occasionally, you'll get an SQLITE_MISUSE that results from failing to properly deal with a previous error, but many times it is a good indication that there is some more basic conceptual misunderstanding about how the library is designed to work.

Extended Codes

The extended codes were added later in the SQLite development cycle. They provide more specific details on the cause of an error. However, because they can change the value returned by a specific error condition, they are turned off by default. You need to explicitly enable them for the older API calls, indicating to the SQLite library that you're aware of the extended error codes and willing to accept them.

All of the standard error codes fit into the least-significant byte of the integer value that is returned by most API calls. The extended codes are all based off one of the standard error codes, but provide additional information in the higher-order bytes. In this way, the extended codes can provide more specific details about the cause of the error. Currently, most of the extended error codes provide specific details for the `SQLITE_IOERR` result. You can find a full list of the extended error codes at *http://sqlite.org/c3ref/c_ioerr_access.html*.

Error Functions

The following APIs are used to enable the extended error codes and extract more information about any current error conditions.

`int sqlite3_extended_result_codes( sqlite3 *db, int onoff )`
: Turns extended result and error codes on or off for this database connection. Database connections returned by any version of `sqlite3_open_xxx()` will have extended codes off by default. You can turn them on by passing a nonzero value in the second parameter. This function always returns `SQLITE_OK`—there is no way to extract the current result code state.

`int sqlite3_errcode( sqlite3 *db )`
: If a database operation returns a non-`SQLITE_OK` status, a subsequent call to this function will return the error code. By default, this will only return a standard error code, but if extended result codes have been enabled, it may also return one of the extended codes.

`int sqlite3_extended_errcode( sqlite3 *db )`
: Essentially the same as `sqlite3_errcode()`, except that extended results are *always* returned.

`const char* sqlite3_errmsg(  sqlite3 *db )`
`const void* sqlite3_errmsg16( sqlite3 *db )`
: Returns a null-terminated, human-readable, English language error string that is encoded in UTF-8 or UTF-16. Any additional calls to the SQLite APIs using this database connection may result in these pointers becoming invalid, so you should either use the string before attempting any other operations, or you should make a private copy. It is also possible for these functions to return a NULL pointer, so check the result value before using it. Extended error codes are not used.

It is acceptable to leave extended error codes off and intermix calls to `sqlite3_errcode()` and `sqlite3_extended_errcode()`.

Because the error state is stored in the database connection, it is easy to end up with race conditionals in a threaded application. If you're sharing a database connection across threads, it is best to wrap your core API call and error-checking code in a critical section. You can grab the database connection's mutex lock with `sqlite3_db_mutex()`. See `sqlite3_db_mutex()` in Appendix G for more details.

Similarly, the error handling system can't deal with multiple errors. If there is an error that goes unchecked, the next call to a core API function is likely to return `SQLITE_MISUSE`, indicating the attempt to use an invalid data structure. In this and similar situations where multiple errors have been encountered, the state of the error message can become inconsistent. You need to check and handle any errors after each API call.

Prepare v2

In addition to the standard and extended codes, the newer `_v2` versions of `sqlite3_prepare_xxx()` change the way prepared statement errors are processed. Although the newer and original versions of `sqlite3_prepare_xxx()` share the same parameters, the `sqlite3_stmt` returned by the `_v2` versions is slightly different.

The most noticeable difference is in how errors from `sqlite3_step()` are handled. For statements prepared with the original version of `sqlite3_prepare_xxx()`, the majority of errors within `sqlite3_step()` will return the rather generic `SQLITE_ERROR`. To find out the specifics of the situation, you had to call `sqlite3_reset()` or `sqlite3_finalize()` to extract a more detailed error code. This would, of course, cause the statement to be reset or finalized, which limited your recovery options.

Things work a bit differently if the statement was prepared with the `_v2` version. In that case, `sqlite3_step()` will return the specific error directly. The call `sqlite3_step()` may return a standard code or an extended code, depending if extended codes are enabled or not. This allows the developer to extract the error directly, and provides for more recovery options.

The other major difference is how database schema changes are handled. If any Data Definition Language command (such as `DROP TABLE`) is issued, there is a chance the prepared statement is no longer valid. For example, the prepared statement may refer to a table or index that is no longer there. The only way to resolve any possible problems is to reprepare the statement.

The `_v2` versions of `sqlite3_prepare_xxx()` make a copy of the SQL statement used to prepare a statement. (This SQL can be extracted. See `sqlite3_sql()` in Appendix G for more details.) By keeping an internal copy of the SQL, a statement is able to reprepare itself if the database schema changes. This is done automatically any time SQLite detects the need to rebuild the statement.

The statements created with the original version of prepare didn't save a copy of the SQL command, so they were unable to recover themselves. As a result, any time the schema changed, an API call involving any previously prepared statement would return `SQLITE_SCHEMA`. The program would then have to reprepare the statement using the original SQL and try again. If the schema change was significant enough that the SQL was no longer valid, `sqlite3_prepare_xxx()` would return an appropriate error when the program attempted to reprepare the SQL command.

Statements created with the `_v2` version of prepare can still return `SQLITE_SCHEMA`. If a schema change is detected and the statement is unable to automatically reprepare itself, it will still return `SQLITE_SCHEMA`. However, under the `_v2` prepare, this is now considered a fatal error, as there is no way to recover the statement.

Here is a side-by-side comparison of the major differences between the original and `_v2` version of prepare:

Statement prepared with original version	Statement prepared with v2 version
Created with `sqlite3_prepare()` or `sqlite3_prepare16()`.	Created with `sqlite3_prepare_v2()` or `sqlite3_prepare16_v2()`.
Most errors in `sqlite3_step()` return `SQLITE_ERROR`.	`sqlite3_step()` returns specific errors directly.
`sqlite3_reset()` or `sqlite3_finalize()` must be called to get full error. Standard or extended error codes may be returned.	No need to call anything additional. `sqlite3_step()` may return a standard or extended error code.
Schema changes will make any statement function return `SQLITE_SCHEMA`. Application must manually finalize and reprepare statement.	Schema changes will make the statement reprepare itself.
If application-provided SQL is no longer valid, the prepare will fail.	If internal SQL is no longer valid, any statement function will return `SQLITE_SCHEMA`. This is a statement-fatal error, and the only choice is to finalize the statement.
Original SQL is not associated with statement.	Statement keeps a copy of SQL used to prepare. SQL can be recovered with `sqlite3_sql()`.
Limited debugging.	`sqlite3_trace()` can be used.

Because the `_v2` error handling is a lot simpler, and because of the ability to automatically recover from schema changes, it is strongly recommended that all new development use the `_v2` versions of `sqlite3_prepare_xxx()`.

Transactions and Errors

Transactions and checkpoints add a unique twist to the error recovery process. Normally, SQLite operates in autocommit mode. In this mode, SQLite automatically wraps each SQL command in its own transaction. In terms of the API, that's the time from when `sqlite3_step()` is first called until `SQLITE_DONE` is returned by `sqlite3_step()` (or when `sqlite3_reset()` or `sqlite3_finalize()` is called).

If each statement is wrapped up in its own transaction, error recovery is reasonably straightforward. Any time SQLite finds itself in an error state, it can simply roll back the current transaction, effectively canceling the current SQL command and putting the database back into the state it was in prior to the command starting.

Once a `BEGIN TRANSACTION` command is executed, SQLite is no longer in autocommit mode. A transaction is opened and held open until the `END TRANSACTION` or `COMMIT TRANSACTION` command is given. This allows multiple commands to be wrapped up in the same transaction. While this is useful to group together a series of discrete commands into an atomic change, it also limits the options SQLite has for error recovery.

When an error is encountered during an explicit transaction, SQLite attempts to save the work and undo just the current statement. Unfortunately, this is not always possible. If things go seriously wrong, SQLite will sometimes have no choice but to roll back the current transaction.

The errors most likely to result in a rollback are `SQLITE_FULL` (database or disk full), `SQLITE_IOERR` (disk I/O error or locked file), `SQLITE_BUSY` (database locked), `SQLITE_NOMEM` (out of memory), and `SQLITE_INTERRUPT` (interrupt requested by application). If you're processing an explicit transaction and receive one of these errors, you need to deal with the possibility that the transaction was rolled back.

To figure out which action was taken by SQLite, you can use the `sqlite3_get_autocommit()` function.

`int sqlite3_get_autocommit( sqlite3 *db )`
: Returns the current commit state. A nonzero return value indicates the database is in autocommit mode, and not in an explicit transaction. A zero value indicates the database is currently inside an explicit transaction.

If SQLite was forced to do a full rollback, the database will once again be in autocommit mode. If the database is not in autocommit mode, it must still be in a transaction, indicating that a rollback was not required.

Although there are situations when it is possible to recover and continue a transaction, it is considered a best practice to always issue a `ROLLBACK` if one of these errors is encountered. In situations when SQLite was already forced to roll back the transaction and has returned to autocommit mode, the `ROLLBACK` will do nothing but return an error that can be safely ignored.

Database Locking

SQLite employs a number of different locks to protect the database from race conditions. These locks allow multiple database connections (possibly from different processes) to access the same database file simultaneously without fear of corruption. The locking system is used for both autocommit transactions (single statements) as well as explicit transactions.

The locking system involves several different tiers of locks that are used to reduce contention and avoid deadlocking. The details are somewhat complex, but the system allows multiple connections to read a database file in parallel, but any write operation requires full, exclusive access to the entire database file. If you want the full details, see *http://www.sqlite.org/lockingv3.html*.

Most of the time the locking system works reasonably well, allowing applications to easily and safely share the database file. If coded properly, most write operations only last a fraction of a second. The library is able to get in, make the required modifications, verify them, and then get out, quickly releasing any locks and making the database available to other connections.

However, if more than one connection is trying to access the same database at the same time, sooner or later they'll bump into each other. Normally, if an operation requires a lock that the database connection is unable to acquire, SQLite will return the error `SQLITE_BUSY` or, in some more extreme cases, `SQLITE_IOERR` (extended code `SQLITE_IOERR_BLOCKED`). The functions `sqlite3_prepare_xxx()`, `sqlite3_step()`, `sqlite3_reset()`, and `sqlite3_finalize()` can all return `SQLITE_BUSY`. The functions `sqlite3_backup_step()` and `sqlite3_blob_open()` can also return `SQLITE_BUSY`, as these functions use `sqlite3_prepare_xxx()` and `sqlite3_step()` internally. Finally, `sqlite3_close()` may return `SQLITE_BUSY` if there are unfinalized statements associated with the database connection, but that's not related to locking.

Gaining access to a needed lock is often a simple matter of waiting until the current holder finishes up and releases the lock. In most cases, this is not a particularly long period of time. The waiting can either be done by the application, which can respond to an `SQLITE_BUSY` by simply trying to reprocess the statement and trying again, or it can be done with a busy handler.

Busy handlers

A busy handler is a callback function that is called by the SQLite library any time it is unable to acquire a lock, but has determined it is safe to try and wait for it. The busy handler can instruct SQLite to keep trying to acquire the lock, or to give up and return an `SQLITE_BUSY` error.

SQLite includes an internal busy handler that uses a timer. If you set a timeout period, SQLite will keep trying to acquire the locks it requires until the timer expires.

`int sqlite3_busy_timeout( sqlite3 *db, int millisec )`
: Sets the given database connection to use the internal timer-based busy handler. If the second parameter is greater than zero, the handler is set to use a timeout value provided in milliseconds (thousandths of a second). If the second parameter is zero or negative, any busy handler will be cleared.

If you want to write your own busy handler, you can set the callback function directly:

`int sqlite3_busy_handler( sqlite3 *db, callback_func_ptr, void *udp )`
: Sets a busy handler for the given database. The second parameter is a function pointer to the busy handler, and the third parameter is a user-data pointer that is passed to the callback. Setting a NULL function pointer will remove the busy handler.

`int user_defined_busy_handler_callback( void *udp, int incr )`
: This is not an SQLite library call, but the format of a user-defined busy handler. The first parameter is the user-data pointer passed to `sqlite3_busy_handler()` when the callback was set. The second parameter is a counter that is incremented each time the busy handler is called while waiting for a specific lock.

  A return value of zero will cause SQLite to give up and return an `SQLITE_BUSY` error, while a nonzero return value will cause SQLite to keep trying to acquire the lock. If the lock is successfully acquired, command processing will continue. If the lock is not acquired, the busy handler will be called again.

Be aware that each database connection has only one busy handler. You cannot set an application busy handler *and* configure a busy timeout value at the same time. Any call to either of these functions will cancel out the other one.

Deadlocks

Setting a busy handler will not fix every problem. There are some situations when waiting for a lock will cause the database connection to deadlock. The deadlock happens when a pair of database connections each have some set of locks and both need to acquire additional locks to finish their task. If each connection is attempting to access a lock currently held by the other connection, both connections will stall in a deadlock. This can happen if two database connections both attempt to write to the database at the same time. In this case, there is no point in both database connections waiting for the locks to be released, since the only way to proceed is if one of the connections gives up and releases all of its locks.

If SQLite detects a potential deadlock situation, it will skip the busy handler and will have one of the database connections return `SQLITE_BUSY` immediately. This is done to encourage the applications to release their locks and break the deadlock. Breaking the deadlock is the responsibility of the application(s) involved—SQLite cannot handle this situation for you.

Avoiding SQLITE_BUSY

When developing code for a system that requires any degree of database concurrency, the easiest approach is to use `sqlite3_busy_timeout()` to set a timeout value that is reasonable for your application. Start with something between 250 to 2,000 milliseconds and adjust from there. This will help reduce the number of `SQLITE_BUSY` response codes, but it will not eliminate them.

The only way to completely avoid SQLITE_BUSY is to ensure a database never has more than one database connection. This can be done by setting PRAGMA locking_mode to EXCLUSIVE.

If this is unacceptable, an application can use transactions to make an SQLITE_BUSY return code easier to deal with. If an application can successfully start a transaction with BEGIN EXCLUSIVE TRANSACTION, this will eliminate the possibility of getting an SQLITE_BUSY. The BEGIN itself may return an SQLITE_BUSY, but in this case the application can simply reset the BEGIN statement with sqlite3_reset() and try again. The disadvantage of BEGIN EXCLUSIVE is that it can only be started when no other connection is accessing the database, including any read-only transactions. Once an exclusive transaction is started, it also locks out all other connections from accessing the database, including read-only transactions.

To allow more concurrency, an application can use BEGIN IMMEDIATE TRANSACTION. If an IMMEDIATE transaction is successfully started, the application is very unlikely to receive an SQLITE_BUSY until the COMMIT statement is executed. In all cases (including the COMMIT), if an SQLITE_BUSY is encountered, the application can reset the statement, wait, and try again. As with BEGIN EXCLUSIVE, the BEGIN IMMEDIATE statement itself can return SQLITE_BUSY, but the application can simply reset the BEGIN statement and try again. A BEGIN IMMEDIATE transaction can be started while other connections are reading from the database. Once started, no new writers will be allowed, but read-only connections can continue to access the database up until the point that the immediate transaction is forced to modify the database file. This is normally when the transaction is committed. If all database connections use BEGIN IMMEDIATE for all transactions that modify the database, then a deadlock is not possible and all SQLITE_BUSY errors (for both the IMMEDIATE writers and other readers) can be handled with a retry.

Finally, if an application is able to successfully begin a transaction of any kind (including the default, DEFERRED), it should never get an SQLITE_BUSY (or risk a deadlock) unless it attempts to modify the database. The BEGIN itself may return an SQLITE_BUSY, but the application can reset the BEGIN statement and try again.

Attempts to modify the database within a BEGIN DEFERRED transaction (or within an autocommit) are the only situations when the database may deadlock, and are the only situations when the response to an SQLITE_BUSY needs to go beyond simply waiting and trying again (or beyond letting the busy handler deal with it). If an application performs modifications within a deferred transaction, it needs to be prepared to deal with a possible deadlock situation.

Avoiding deadlocks

The rules to avoid deadlocks are fairly simple, although their application can cause significant complexity in code.

First, the easy ones. The functions `sqlite3_prepare_xxx()`, `sqlite3_backup_step()`, and `sqlite3_blob_open()` cannot cause a deadlock. If an `SQLITE_BUSY` code is returned from one of these functions at any time, simply wait and call the function again.

If the function `sqlite3_step()`, `sqlite3_reset()`, or `sqlite3_finalize()` returns `SQLITE_BUSY` from within a deferred transaction, the application must back off and try again. For statements that are not part of an explicit transaction, the prepared statement can simply be reset and re-executed. For statements that are inside an explicit deferred transaction, the whole transaction must be rolled back and started over from the beginning. In most cases, this will happen on the first attempt to modify the database. Just remember that the whole reason for the rollback is that some other database connection needs to modify the database. If an application has done several read operations to prepare a write operation, it would be best to reread that information in a new transaction to confirm the data is still valid.

Whatever you do, don't ignore `SQLITE_BUSY` errors. They can be rare, but they can also be a source of great frustration if handled improperly.

When BUSY becomes BLOCKED

When a database connection needs to modify the database, a lock is placed that makes the database read-only. This allows other connections to continue to read the database, but prevents them from making modifications. The actual changes are held in the database page cache and not yet written to the database file. Writing the changes out would make the changes visible to the other database connections, breaking the isolation rule of transactions. Since the changes have not yet been committed, it is perfectly safe to have them cached in memory.

When all the necessary modifications have been made and the transaction is ready to commit, the writer further locks the database file so that new read-only transactions cannot get started. This allows the existing readers to finish up and release their own database locks. When all readers are finished, the writer should have exclusive access to the database and may finally flush the changes out of the page cache and into the database file.

This process allows read-only transactions to continue running while the write transaction is in progress. The readers need to be locked out only when the writer actually commits its transaction. However, a key assumption in this process is that the changes fit into the database page cache and do not need to be written until the transaction is committed. If the cache fills up with pages that contain pending changes, a writer has no option but to put an exclusive lock on the database and flush the cache prior to the commit stage. The transaction can still be rolled back at any point, but the writer must be given immediate access to the exclusive write lock in order to perform the cache flush.

If this lock is not immediately available, the writer is forced to abort the entire transaction. The write transaction will be rolled back and the extended result code `SQLITE_IOERR_BLOCKED` (standard code `SQLITE_IOERR`) will be returned. Because the transaction is automatically rolled back, there aren't many options for the application, other than to start the transaction over.

To avoid this situation, it is best to start large transactions that modify many rows with an explicit `BEGIN EXCLUSIVE`. This call may fail with `SQLITE_BUSY`, but the application can simply retry the command until it succeeds. Once an exclusive transaction has started, the write transaction will have full access to the database, eliminating the chance of an `SQLITE_IOERR_BLOCKED`, even if the transaction spills out of the cache prior to the commit. Increasing the size of the database cache can also help.

Utility Functions

The SQLite library contains a number of utility functions that are useful for both application developers, and those working on SQLite extensions. Most of these are not required for basic database tasks, but if your code is strongly tied to SQLite, you may find these particularly useful.

Version Management

There are several functions available to query the version of the SQLite library. Each API call has a corresponding `#define` macro that declares the same value.

`SQLITE_VERSION`
`const char* sqlite3_libversion( )`
: Returns the SQLite library version as a UTF-8 string.

`SQLITE_VERSION_NUMBER`
`int sqlite3_libversion_number( )`
: Returns the SQLite library version as an integer. The format is *`MNNNPPP`*, where *`M`* is the major version (3, in this case), *`N`* is the minor number, and *`P`* is the point release. This format allows for releases up to 3.999.999. If a sub-point release is made, it will not be indicated in this version number.

`SQLITE_SOURCE_ID`
`const char* sqlite3_sourceid( )`
: Returns the check-in stamp of the code used in this release. The string consists of a date, time stamp, and an SHA1 hash of the source from the source repository.

If you're building your own application, you can use the `#define` macros and the function calls to verify that you're using the correct header for the available library. The `#define` values come from the header file, and are set when your application is compiled. The function calls return the same values that were baked into the library when it was compiled.

If you're using a dynamic library of some sort, you can use these macros and functions to prevent your application from linking with a library version other than the one it was originally compiled against. This might not be a good thing, however, as it will also prevent upgrades. If you need to lock in a specific version, you should probably be using a static library.

If you want to check the validity of a dynamic library, it might be better to do something like this:

```
if ( SQLITE_VERSION_NUMBER > sqlite3_libversion_number( ) ) {
    /* library too old; report error and exit. */
}
```

Remember that the macro will hold the version used when your code was compiled, while the function call will return the version of the SQLite library. In this case, we report an error if the library is older (smaller version) than the one used to compile and build the application code.

Memory Management

When SQLite needs to dynamically allocate memory, it normally calls the default memory handler of the underlying operating system. This causes SQLite to allocate its memory from the application heap. However, SQLite can also be configured to do its own internal memory management (see `sqlite3_config()` in Appendix G). This is especially important on embedded and hand-held devices where memory is limited and overallocation may lead to stability problems.

Regardless, you can access whatever memory manager SQLite is using with these SQLite memory management functions:

`void* sqlite3_malloc( int numBytes )`
: Allocates and returns a buffer of the size specified. If the memory cannot be allocated, a NULL pointer is returned. Memory will always be 8-byte (64-bit) aligned.

  This is a replacement for the standard C library `malloc()` function.

`void* sqlite3_realloc( void *buffer, int numBytes )`
: Used to resize a memory allocation. Buffers can be made larger or smaller. Given a buffer previously returned by `sqlite3_malloc()` and a byte count, `*_realloc()` will allocate a new buffer of the specified size and copy as much of the old buffer as will fit into the new buffer. It will then free the old buffer and return the new buffer. If the new buffer cannot be allocated, a NULL is returned and the original buffer is *not* freed.

  If the buffer pointer is NULL, the call is equivalent to a call to `sqlite3_malloc()`. If the `numBytes` parameter is zero or negative, the call is equivalent to a call to `sqlite3_free()`.

  This is a replacement for the standard C library `realloc()` function.

`void sqlite3_free( void *buffer )`

Releases a memory buffer previously allocated by `sqlite3_malloc()` or `sqlite3_realloc()`. Also used to free the results or buffers of a number of SQLite API functions that call `sqlite3_malloc()` internally.

This is a replacement for the standard C library `free()` function.

While a number of SQLite calls require the use of `sqlite3_free()`, application code is free to use whatever memory management is most appropriate. Where these functions become extremely useful is in writing custom functions, virtual tables, or any type of loadable module. Since this type of code is meant to operate in any SQLite environment, you will likely want to use these memory management functions to ensure the maximum portability for your code.

Summary

With the information in this chapter, you should be able to write code that opens a database file and executes SQL commands against it. With that ability, and the right SQL commands, you should be able to create new databases, specify tables, insert data, and build complex queries. For a large number of applications, this is enough to service most of their database needs.

The next chapters look at more advanced features of the SQLite C API. This includes the ability to define your own SQL functions. That will enable you to extend the SQL used by SQLite with simple functions, as well as aggregators (used with `GROUP BY`) and sorting collations. Additional chapters will look at how to implement virtual tables and other advanced features.