CHAPTER 5

Design and Concepts

This chapter sets the stage for the next three chapters, which are exclusively devoted to programming with SQLite. It addresses the things that you as a programmer should know about SQLite when using it in your code. Whether you are programming with SQLite in its native C, or in your favorite scripting language, it helps to understand not only its API, but also a little about its architecture and implementation. Armed with this knowledge, you will be better equipped to write code that runs faster and avoids potential pitfalls such as deadlocks or unexpected errors. You will see how SQLite works in relation to your code, and you can be more confident that you are attacking the problem from the right direction.

You don't have to comb through the bowels of the source to understand these things, nor do you have to be a C programmer. SQLite's design and concepts are all very straightforward and easy to understand. And there are only a few things you need to know. This chapter lays them out for you.

Obviously, you need to know how the API works. So this chapter starts with a conceptual introduction to the API, illustrating its major data structures, its general design, as well as its major functions. It also looks at some of the major SQLite subsystems that play important roles in query processing.

Beyond just knowing what functions do what, you also need look above the API, seeing how everything operates in terms of transactions. Everything involving a SQLite database is done within the context of a transaction. Then you need to look beneath the API, to see how transactions work in terms of locks. Locks can cause problems if you don't know how they operate. By understanding locks, not only can you avoid potential concurrency problems, you can also optimize your queries by controlling how your program uses them.

Finally, you have to understand how all of these things apply to writing code. The last part of the chapter brings all three topics—the API, transactions, and locks—together and looks at different examples of good and bad code. It specifically identifies scenarios that could cause problems and provides some insight on how to address them.

With these things in order, you will be well on your way to conquering the C API, or the API of any language extension.

The API

Functionally, the SQLite API can be separated into two general parts: the core API and extension API. The core API consists of all the functions used to perform basic database operations: connecting to the database, processing SQL, and iterating through results. It also includes

various utility functions that help with tasks such as string formatting, operational control, debugging, and error handling. The extension API offers different ways to extend SQLite by creating your own user-defined SQL extensions, which you can integrate into SQLite's SQL dialect.

What's New in SQLite Version 3

Before we get started, let's talk about some of the features that have been added to SQLite version 3, as it introduced many major improvements to all major subsystems.

To begin with, SQLite's API has been completely redesigned, and has many new features. It grew from approximately 15 functions in version 2 to 88 functions in version 3. The new API includes native support for UTF-8 and UTF-16 encodings along with many new utility functions. It has a more flexible query model that makes prepared queries easier and supports new parameter binding methods. It also added user-defined collating sequences, `CHECK` constraints, 64-bit key values, and a new query optimizer.

The backend has dramatically improved concurrency and performance. It has a new locking system that introduces a lock escalation model. This system solves the writer starvation problem in SQLite 2, where new readers can continually prevent writers from getting exclusive access to the database. The new model ensures that writers gain exclusive access on a first-come, first-served basis. Furthermore, writers can even begin performing work before acquiring exclusive access to the database by storing changes in a temporary buffer. The new model has been shown to increase performance for write-intensive applications by as much as 400 percent over version 2.

SQLite 3 includes an improved B-tree module, which now uses B+-trees for tables. B+-trees yield better overall search performance, store larger data fields more efficiently, and omit unused fields from disk. As a result, database files are typically 25–35 percent smaller and offer better overall performance. The B-tree module is covered in more detail in Chapter 9.

SQLite 3 significantly changed its storage model. It went from text-only in version 2 to a new model that supports five native data types, in addition to manifest typing and type affinity as explained in Chapter 4. Each type is optimized for better overall search performance and also uses less storage space. Integer and floating point values, for example, are stored in binary format rather than ASCII and therefore don't have to be converted for evaluation in `WHERE` clause expressions, as in version 2. Manifest typing gives you the option of declaring a column's types in a meaningful way or not at all. Type affinity determines the format in which a value is stored in a column, based on the value's representation and the column's affinity. Type affinity is directly related to manifest typing—a column's affinity is determined by its declared type, or lack thereof.

In many ways, SQLite version 3 is a completely different database from SQLite version 2 and offers numerous advantages in flexibility, features, and performance.

The Principal Data Structures

As you saw in Chapter 1, there are many components in SQLite—parser, tokenizer, virtual machine, etc. But from a programmer's point of view, the main things to know about are connections, statements, the B-tree, and the pager. Their relationships to one another are shown in Figure 5-1. These objects collectively address the three principal things you must know about SQLite to write good code: the API, transactions, and locks.

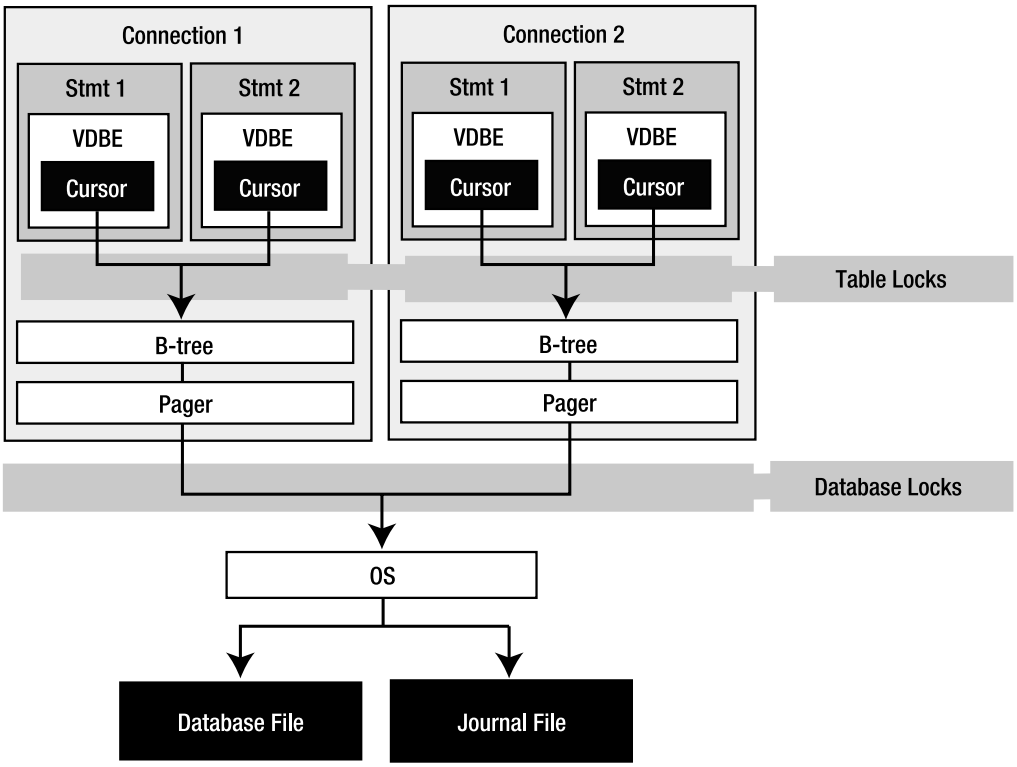


Figure 5-1. *SQLite C API object model*

Technically, the B-tree and pager are not part of the API; they are off limits. But they play a critical role in transactions and locks. We will explore their involvement in these matters here in this section and later in the section "Transactions."

Connections and Statements

The two fundamental data structures in the API associated with query processing are the connection and the statement. In most language extensions you will see both a connection object and a statement object, which are used to execute queries. In the C API, they correspond directly to the `sqlite3` and the `sqlite3_stmt` handles, respectively. Every major operation in the API is done using one of these two structures.

A connection represents a single connection to a database as well as a single transaction context. Statements are derived from connections. That is, every statement has an associated connection object. A statement represents a single compiled SQL statement. Internally, it is expressed in the form of VDBE byte code—a program that when executed will carry out the SQL command. Statements contain everything needed to execute a command. They include resources to hold the state of the VDBE program as it is executed in a stepwise fashion, B-tree cursors that point to records on disk, and other things such as bound parameters, which are addressed later in the section "Parameter Binding." While they contain many different things,

you can simply think of them as cursors with which to iterate through a result set, or as opaque handles referencing a single SQL command.

The B-tree and Pager

Each connection can have multiple database objects—one main database followed by any attached databases. Each database object has one B-tree object, which in turn has one pager object.

Statements use their connection's B-tree and pager objects to read and write data to and from the database. Statements that read the database iterate over B-trees using cursors. Cursors iterate over records, and records are stored in pages. As a cursor traverses records, it also traverses pages. For a cursor to access a page, it must first be loaded from disk into memory. This is the pager's job. Whenever the B-tree needs a particular page in the database, it asks the pager to fetch it from disk. The pager then loads the page into its page cache, which is a memory buffer. Once it is in the page cache, the B-tree and its associated cursor can get to the records inside the page.

If the cursor modifies the page, then the pager must also take special measures to preserve the original page in the event of a transaction rollback. Thus, the pager is responsible for reading and writing to and from the database, maintaining a memory cache or pages, as well as managing transactions. In addition to this, it manages locks and crash recovery. All of these responsibilities are covered later in "Transactions."

There are two things you should know about connections and transactions in general. First, when it comes to any operation on the database, a connection *always* operates under a transaction. Second, a connection *never* has more than one transaction open at a time. Basically, whenever a connection does anything with a database, it always operates under exactly one transaction, no more, no less. You can't avoid it.

Therefore, all statements derived from a given connection operate within the same transaction context. So if you allocate two statements from the same connection, and the first issues a `BEGIN` and the second issues a `COMMIT`, then they collectively completed a single transaction. The first statement started it and the second ended it. If you want the two statements to run in separate transactions, then you have to use multiple connections—one connection for each transaction context.

The Core API

As mentioned earlier, the core API is concerned with executing SQL commands. It is made of various functions for performing queries as well as various utility functions for managing other aspects of the database. There are two essential methods for executing SQL commands: prepared queries and wrapped queries. Prepared queries are the way in which SQLite ultimately executes all commands, both in the API and internally. It is a three-phase process consisting of preparation, execution, and finalization. There is a single API function associated with each phase. Associated with the execution phase are functions with which to obtain record and column information from result sets.

In addition to the standard query method, there are two wrapper functions, which wrap the three phases into a single function call. They provide a convenient way to execute a SQL command all at once. These functions are just a few of the many miscellaneous utility functions in the API. We will look at all of the query methods along with their associated utility functions in this section. Before we do however, let's first look at how to connect to a database.

The Connection Lifecycle

Connections are the sole means through which to operate on a database. The connection lifecycle consists of three phases:

1. **Connect to the database**. Connecting involves creating a database connection object of some sort. The connection object manages transactions (through its associated pager), often performs simple one-step queries (using the wrapped query functions), and creates statement objects for prepared queries.
2. **Perform transactions**. As you know, all commands are executed within transactions. By default, a database connection runs in autocommit mode. This means that every SQL command it executes runs under its own independent transaction. The alternative is to manually declare transactions using `BEGIN..COMMIT`. In this scenario, multiple commands can run together within the same transaction.
3. **Disconnect from the database**. Disconnecting from a database involves closing the database file and the files of any attached databases.

Other activities involved with query processing include handling errors, busy conditions, and schema changes, all of which are done through the utility functions covered in the following sections.

Connecting to a Database

Connecting to a database involves little more than opening a file. Every SQLite database is stored in a single operating system file—one database to one file. The function used to connect, or open, a database in the C API is `sqlite3_open()`, and is basically just a system call for opening a file. SQLite can also create in-memory databases. In most extensions, if you use `:memory:` or an empty string as the name for the database, it will create the database in RAM. The database will only be accessible to the connection that created it (it cannot be shared with other connections). Furthermore, the database will only last for the duration of the connection. It is deleted from memory when the connection closes.

When you connect to a database on disk, SQLite opens a file, if it exists. If you try to open a file that doesn't exist, SQLite will assume that you want to create a new database. In this case, SQLite doesn't immediately create a new operating system file. It will only create a new file if you put something into the new database—create a table or view or other database object. If you just open a new database, do nothing, and close it, SQLite does not bother with creating a database file—it would just be an empty file anyway.

There is also another reason for not creating a new file right away. Certain database options, such as encoding, page size, and autovacuum, can only be set before you create a database. By default, SQLite uses a 1,024-byte page size. However, you can use different page sizes ranging from 512 to 32,768 bytes by powers of 2. You might want to use different page sizes for performance reasons. For example, setting the page size to match the operating system's page size can sometimes make I/O more efficient. Sometimes larger page sizes help with applications that deal with a lot of binary data. You set the database page size using the `page_size` pragma.

Encoding is another permanent database setting. You specify a database's encoding using the `encoding` pragma, which can be `UTF-8`, `UTF-16`, `UTF-16le` (little endian), and `UTF-16be` (big endian).

Finally there is autovacuum, which you set with the `auto_vacuum` pragma. When a transaction deletes data from a database, SQLite's default behavior is to keep the deleted pages around for recycling. The database file remains the same size. To free the pages, you must explicitly issue a `VACUUM` command to reclaim the unused space. The autovacuum feature causes SQLite to automatically shrink the database file when data is deleted. This feature is often more useful in embedded applications where storage is a premium.

Once you open a database—file or memory—it will be represented internally by an opaque `sqlite3` connection handle. This handle represents a single connection to a database. Connection objects in extensions abstract this handle, and sometimes implement methods that correspond to API functions that take the handle as an argument.

Executing Prepared Queries

As stated earlier, the prepared query method is the actual process by which SQLite executes all SQL commands. Executing a SQL command is a three-step process:

- **Preparation**: The parser, tokenizer, and code generator prepare the SQL statement by compiling it into VDBE byte code. In the C API, this is performed by the `sqlite3_prepare()` function, which talks directly to the compiler. The compiler creates a `sqlite3_stmt` handle (statement handle) that contains the byte code and all other resources needed to execute the command and iterate over the result set (if the command produces one).

- **Execution**: The VDBE executes the byte code. Execution is a stepwise process. In the C API, each step is initiated by `sqlite3_step()`, which causes the VDBE to step through the byte code. The first call to `sqlite3_step()` usually acquires a lock of some kind, which varies according to what the command does (reads or writes). For `SELECT` statements, each call to `sqlite3_step()` positions the statement handle's cursor on the next row of the result set. For each row in the set, it returns `SQLITE_ROW` until it reaches the end, whereupon it returns `SQLITE_DONE`. For other SQL statements (`INSERT`, `UPDATE`, `DELETE`, etc.), the first call to `sqlite3_step()` causes the VDBE to process the entire command.

- **Finalization**: The VDBE closes the statement and deallocates resources. In the C API, this is performed by `sqlite3_finalize()`, which causes the VDBE to terminate the program and close the statement handle. However, if a transaction is manually started, it must be manually committed or rolled back, or `sqlite3_finalize()` will return an error. When `sqlite3_finalize()` is successful, all resources associated with the statement object are freed. In autocommit mode, it also releases the associated database lock.

Each step—preparation, execution, finalization—corresponds to a respective statement handle state—prepared, active, or finalized. Prepared means that all necessary resources have been allocated and the statement is ready to be executed, but nothing has been started. No lock has been acquired, nor will a lock be acquired until the first call to `sqlite3_step()`. The active state starts with the first call to `sqlite3_step()`. At that point the statement is in the process of being executed and some kind of lock is in play. Finalized means that the statement is closed and all associated resources have been freed. These steps and states are illustrated in Figure 5-2.

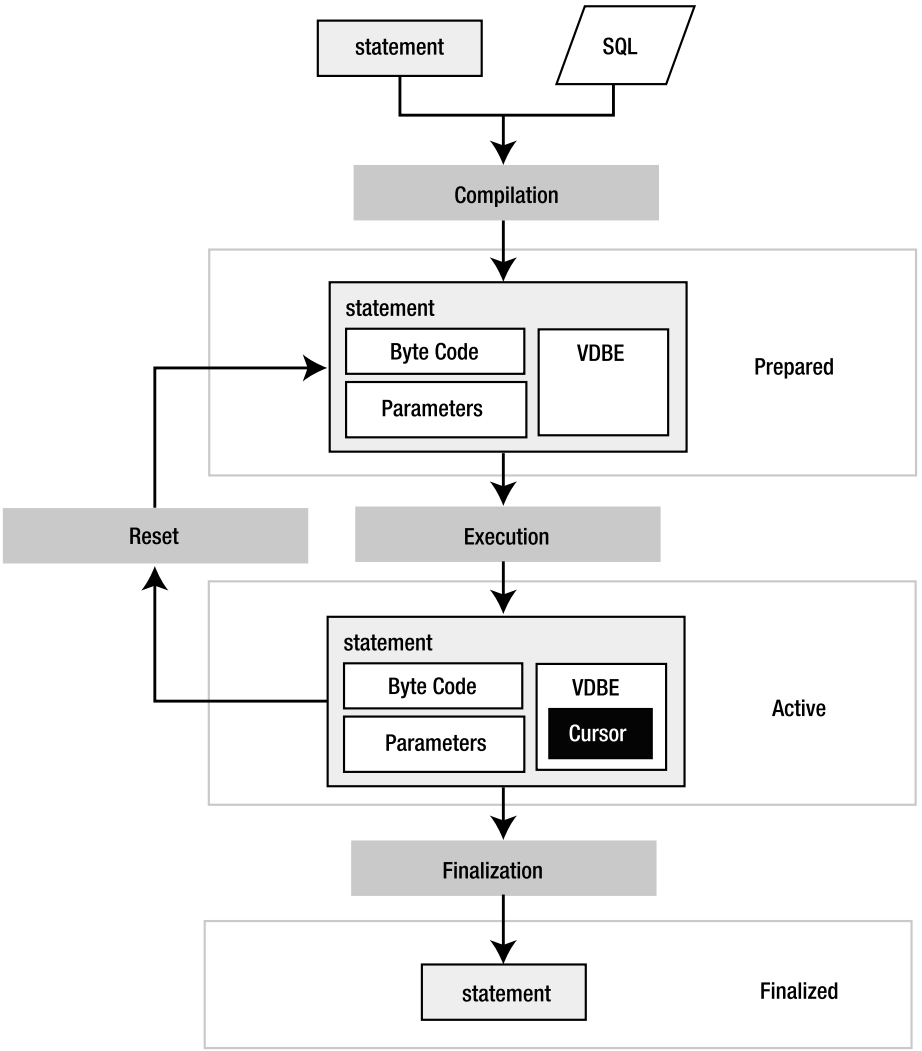


Figure 5-2. *Statement processing*

The following pseudocode illustrates the general process of executing a query in SQLite:

```
# 1. Open the database, create a connection object (db)
db = open('foods.db')

# 2.A. Prepare a statement
stmt = db.prepare('SELECT * FROM episodes')

# 2.B. Execute. Call step() is until cursor reaches end of result set.
while stmt.step() == SQLITE_ROW
  print stmt.column('name')
end
```

```
# 2.C. Finalize. Release read lock.
stmt.finalize()

# 3. Insert a record
stmt = db.prepare('INSERT INTO foods VALUES (…)')
stmt.step()
stmt.finalize()

# 4. Close database connection.
db.close()
```

This pseudocode is an object-oriented analog of the API, similar to what you might find in a scripting language. The methods all correspond to SQLite API functions. For example, `prepare()` mirrors `sqlite3_prepare()`, and so on. This example performs a `SELECT`, iterating over all returned rows, followed by an `INSERT`, which is processed by a single call to `step()`.

TEMPORARY STORAGE

Temporary storage is an important part of query processing. SQLite occasionally needs to store intermediate results produced in the process of executing commands—for instance, when results need to be sorted for an `ORDER BY` clause, or rows in one table are joined with rows in another table. This information is often stored in temporary storage. Temporary storage is kept either in RAM or in a file. While SQLite has suitable defaults for all platforms, you may want to control how and where it uses this storage. The `temp_store` pragma lets you specify whether to use RAM or file-based storage. If you use file-based storage, you can use the `temp_store_directory` pragma to specify where the storage file is created.

Using Parameterized SQL

SQL statements can contain parameters. Parameters are placeholders in which values may be provided (or "bound") at a later time after compilation. The following statements are examples of parameterized queries:

```
INSERT INTO foods (id, name) VALUES (?,?);
INSERT INTO episodes (id, name) (:id, :name);
```

These statements represent two forms of parameter binding: *positional* and *named*. The first command uses positional parameters and the second command uses named parameters.

Positional parameters are defined by the position of the question mark in the statement. The first question mark has position 1, the second 2, and so on. Named parameters use actual variable names, which are prefixed with a colon. When `sqlite3_prepare()` compiles a statement with parameters, it allocates placeholders for the parameters in the resulting statement handle. It then expects values to be provided for these parameters before the statement is executed. If you don't bind a value to a parameter, SQLite will use `NULL` as the default when it executes the statement.

The advantage of parameter binding is that you can execute the same statement multiple times without having to recompile it. You just reset the statement, bind a new set of values,

and reexecute. This is where resetting rather than finalizing a statement comes in handy: it avoids the overhead of SQL compilation. By resetting a statement, you are reusing the compiled SQL code. You completely avoid the tokenizing, parsing, and code generation overhead. Resetting a statement is implemented in the API by the `sqlite3_reset()` function.

The other advantage of parameters is that SQLite takes care of escaping the string values you bind to parameters. For example, if you had a parameter value such as `'Kenny's Chicken'`, the parameter binding process will automatically convert it to `'Kenny''s Chicken'`—escaping the single quote for you, helping you avoid syntax errors and possible SQL injection attacks (covered in the section "Formatting SQL Statements"). The following pseudocode illustrates the basic process of using bound parameters:

```
db = open('foods.db')
stmt = db.prepare('INSERT INTO episodes (id, name) VALUES (:id, :name)')

stmt.bind('id', '1')
stmt.bind('name', 'Soup Nazi')
stmt.step()

# Reset and use again
stmt.reset()
stmt.bind('id', '2')
stmt.bind('name', 'The Junior Mint')

# Done
stmt.finalize()

db.close()
```

Here, `reset()` simply deallocates the statement's resources, but leaves its VDBE byte code and parameters intact. The statement is ready to run again without the need for another call to `prepare()`. This can significantly improve performance of repetitive queries such as this because the compiler completely drops out of the equation.

Executing Wrapped Queries

As mentioned earlier, there are two very useful utility functions that wrap the prepared query process, allowing you to execute SQL commands in a single function call. One function—`sqlite3_exec()`—is typically for queries that don't return data. The other—`sqlite3_get_table()`—is typically for queries that do. In many language extensions you will see analogs to both functions. Most extensions refer to the first method simply as `exec()`, and the second as just `get_table()`.

The `exec()` function is a quick and easy way to execute `INSERT`, `UPDATE`, and `DELETE` statements or DDL statements for creating and destroying database objects. It works straight from the database connection, taking a `sqlite3` handle to an open database along with a string containing one or more SQL statements. That's right; `exec()` is capable of processing a string of *multiple* SQL statements delimited by semicolons and running them all together.

Internally, `exec()` parses the SQL string, identifies individual statements, and then processes them one by one. It allocates its own statement handles and prepares, executes, and finalizes

each statement. If multiple statements are passed to it and one of them fails, exec() terminates execution on that command, returning the associated error code. Otherwise, it returns a success code. The following pseudocode illustrates conceptually how exec() works in an extension:

```
db = open('foods.db')
db.exec("INSERT INTO episodes (id, name) VALUES (1, 'Soup Nazi')")
db.exec("INSERT INTO episodes (id, name) VALUES (2, 'The Fusilli Jerry')")
db.exec("BEGIN; DELETE from episodes; ROLLBACK")
db.close()
```

While you can also use exec() to process records returned from SELECT, it involves subtle methods for doing so that are generally supported only by the C API.

The second query function, sqlite3_get_table(), is somewhat of a misnomer as it is not restricted to just querying a single table. Rather, its name refers to the tabular results of a SELECT query. You can certainly process joins with it just as well. In many respects, get_table() works in the same way as exec(), but it returns a complete result set in memory. This result set is represented in various ways depending on the extension. The following pseudocode illustrates how it is typically used:

```
db = open('foods.db')
table = db.get_table("SELECT * FROM episodes LIMIT 10")

for i=0; i < table.rows; i++
    for j=0; j < table.cols; j++
        print table[i][j]
    end
end

db.close()
```

The upside of get_table() is that it provides a one-step method to query and get results. The downside is that it stores the results completely in memory. So the larger the result set, the more memory it consumes. Not surprisingly, then, it is not a good idea to use get_table() to retrieve large result sets. The prepared query method, on the other hand, only holds one record (actually its associated database page) in memory at a time, so it is much better suited for traversing large result sets.

Notice that while these functions buy you convenience, you also lose a bit of control simply by not having access to a statement handle. For example, you can't use parameterized SQL statements with either of them. So they are not going to be as efficient for repetitive tasks that could benefit from parameters. Also, the API includes functions that work with statement handles that provide lots of information about columns in a result set—both data and metadata. These are not available with wrapped queries either. Wrapped queries have their uses, to be sure. But prepared queries do as well.

Handling Errors

The previous examples are greatly oversimplified to illustrate the basic parts of query processing. In real life, you always have to consider the possibility of errors. Almost every function you have seen so far can encounter errors of some sort. Common error codes you need to be prepared to

handle include SQLITE_ERROR, SQLITE_BUSY, and SQLITE_SCHEMA. The latter two error codes refer to busy conditions that arise when either a connection can't get a lock or the database schema changes between the time a statement is compiled and executed. Busy conditions are addressed in the "Transactions" section while schema errors are covered in detail in Chapter 6.

Many language extensions often handle schema errors differently. Some transparently report them as busy conditions. Others return the actual error code. In any case, when you encounter a schema error, it means that some other connection has changed the database schema and your current statement is no longer valid. You simply have to recompile the statement in order to execute it. Schema errors can only occur between a call to prepare() and the *first* call to step(). If your first call to step() succeeds, then you do not have to worry about schema errors for subsequent calls to step(), as your connection has a lock on the database that prevents other connections from changing the database schema during that time.

With regard to general errors, the API provides the return code of the last called function with the sqlite3_errcode() function. You can get more specific error information using the sqlite3_errmsg() function, which provides a text description of the last error. Most language extensions support this function in some way or another.

With this in mind, each call in the previous example should check for the appropriate errors using something like the following:

```
# Check and report errors
if db.errcode() != SQLITE_SUCCESS
    print db.errmsg(stmt)
end
```

In general, error handling is not difficult. The way you handle any error depends on what exactly you are trying to do. The easiest way to approach error handling is the same as with any other API—read the documentation on the function you are using and code defensively.

Formatting SQL Statements

Another nice convenience function you may see some extensions support is sqlite3_mprintf(). It is a variant of the standard C library sprintf(). It has special substitutions that are specific to SQL that can be very handy. These substitutions are denoted %q and %Q, and escape SQL-specific values. %q works like %s in that it substitutes a null-terminated string from the argument list. But it also doubles every single-quote character as well as every backslash, making your life easier and helping guard against SQL injection attacks (see the sidebar "SQL Injection Attacks"). For example:

```
char* before = "Hey, at least %q no pig-man.";
char* after = sqlite3_mprintf(before, "\he's\");
```

The value after produced here is 'Hey, at least \\he''s\\ no pig-man'. The single quote in *he's* is doubled along with the backslashes around it, making it acceptable as a string literal in a SQL statement. The %Q formatting does everything %q does, but it additionally encloses the resulting string in single quotes. Furthermore, if the argument for %Q is a NULL pointer (in C), it produces the string NULL without single quotes. For more information, see the sqlite3_mprintf() documentation in the C API reference in Appendix B.

SQL INJECTION ATTACKS

If your application relies on any user input with which to construct SQL statements, you could be vulnerable to a SQL injection attack. If you are not careful to filter user input, it could be possible for someone to craft input that could alter the SQL statement, injecting a new SQL statement into the string. For example, say your program uses user input to fill in the value of the following SQL statement:

```
SELECT * FROM foods WHERE name='%s';
```

You replace the `%s` with whatever the user supplies. If the user has any knowledge of your database, he or she could provide input that can dramatically alter the SQL statement. For example, say the user were to provide the following string value for the `name` input:

```
nothing' LIMIT 0; SELECT name FROM sqlite_master WHERE name='%
```

After substituting the user's input into your SQL statement, the new statement turns into two statements:

```
SELECT * FROM foods WHERE name='nothing' LIMIT 0; SELECT name FROM
sqlite_master WHERE name='%';
```

The first statement will return nothing and the second will return the names of all objects in your database. Granted, the odds of this happening require quite a bit of knowledge on the attacker's part, but it is nevertheless possible. Some major (commercial) web applications have been known to keep SQL statements embedded in their JavaScript, which can provide plenty of hints about the database being used. In the previous example, all a malicious user has to do now is insert `DROP TABLE` statements for every table found in `sqlite_master` and you could find yourself fumbling through backups.

Operational Control

The API includes a variety of commands that allow you to monitor, control, or generally limit what can happen in a database. SQLite implements them in the form of filters, or callback functions that you can register to be called for specific events. There are three "hook" functions: `sqlite3_commit_hook()`, which monitors transaction commits on a connection; `sqlite3_rollback_hook()`, which monitors rollbacks; and `sqlite3_update_hook()`, which monitors changes to rows from `INSERT`, `UPDATE`, and `DELETE` operations. These hooks are called at runtime—while a command is executed. Each hook allows you to register a callback function on a connection-by-connection basis, and lets you provide some kind of application-specific data to be passed to the callback as well. The general use of operational control functions is as follows:

```
def commit_hook(cnx)
  log('Attempted commit on connection %x', cnx)
  return -1
end
db = open('foods.db')
db.set_commit_hook(rollback_hook, cnx)
db.exec("BEGIN; DELETE from episodes; ROLLBACK")
db.close()
```

A hook's return value has the power to alter the event in specific ways, depending on the hook. In this example, because the commit hook returns a non-zero value, the commit will be rolled back.

Additionally, the API provides a very powerful compile time hook called `sqlite3_set_authorizer()`. This function provides you with fine-grained control over almost everything that happens in the database as well as the ability to limit both access and modification on a database, table, and column basis. This function is covered in detail in Chapter 6.

Using Threads

SQLite has a number of functions for using it in a multithreaded environment. With version 3.3.1, SQLite introduced a unique operational mode called *shared cache mode*, which is designed for multithreaded embedded servers. This model provides a way for a single thread to host multiple connections that share a common page cache, thus lowering the overall memory footprint of the server. It also employs a different concurrency model. Included with this feature are various functions for managing memory and fine-tuning the server. This operational mode is explained further later in the section "Shared Cache Mode," and in full detail in Chapter 6.

The Extension API

The extension API in the SQLite C API offers support for user-defined functions, aggregates, and collations. A user-defined function is a SQL function that maps to some handler function that you implement in C or another language. When using the C API, you implement this handler in C or C++. In language extensions, you implement the handler in the same language as the extension.

User-defined extensions must be registered on a connection-by-connection basis as they are stored in program memory. That is, they are *not* stored in the database, like stored procedures in larger relational database systems. They are stored in your program. When your program or script starts up, it is responsible for registering the desired user-defined extensions for each connection that it intends to use them.

Creating User-Defined Functions

Implementing a user-defined function is a two-step process. First, you write the handler. The handler does something that you want to perform from SQL. Next, you register the handler, providing its SQL name, its number of arguments, and a pointer (or reference) to the handler.

For example, say you wanted to create a special SQL function called `hello_newman()`, which returns the text `'Hello Jerry'`. In the SQLite C API, you would first create a C function to implement this, such as the following:

```
void hello_newman(sqlite3_context* ctx, int nargs, sqlite3_value** values)
{
    /* Create Newman's reply */
    const char *msg = "Hello Jerry";

    /* Set the return value. Have sqlite clean up msg w/ sqlite_free(). */
    sqlite3_result_text(ctx, msg, strlen(msg), sqlite3_free);
}
```

Don't worry if you don't know C or the C API. This handler just returns `'Hello Jerry'`. Next, to actually use it, you register this handler using the `sqlite3_create_function()` (or the equivalent function in your language):

```
sqlite3_create_function(db, "hello_newman", 0, hello_newman);
```

The first argument (`db`) is the database connection. The second argument is the name of the function as it will appear in SQL, and the third argument means that the function takes 0 arguments. (If you provide -1, then it means that the function accepts a variable number of arguments.) The last argument is a pointer to the `hello_newman()` C function, which will be called when the SQL function is called.

Once registered, SQLite knows that when it encounters the SQL function `hello_newman()`, it needs to call the C function `hello_newman()` to obtain the result. Now, you can execute `SELECT hello_newman()` within your program and it will return a single row with one column containing the text `'Hello Jerry'`.

As mentioned, many language extensions allow you to implement user-defined functions in their respective language. For example, the Java interface provides you with the means to implement `hello_newman()` in Java, Perl in Perl, and likewise in many other extensions. Different extensions register user-defined functions in different ways, sometimes employing specific features of their respective languages to do so. In Ruby, for example, you implement a user-defined function using a *block*—one of its particular language constructs. Consider the following Ruby program, which creates a user-defined function called `bool()`:

```
db = SQLite3::Database.new(':memory:')

db.create_function('bool', 1) do |func, *args|
  if args[0].to_s.upcase == 'TRUE' or args[0].to_s.upcase == 'FALSE'
    func.result = 1
  else
    func.result = 0
  end
end

def execute(db, sql)
  begin
    db.query(sql)
  rescue Exception
    puts "Statement failed: #{$!}"
  end
end

execute(db, 'create table domain (x CHECK(bool(x)))')
execute(db, "insert into domain values ('true')")
execute(db, "insert into domain values ('JujyFruit')")
```

The `bool()` function takes a single argument and returns 1 (true) if the argument meets its definition of a Boolean value or 0 (false) if not. The code then uses this function in a `CHECK` constraint to impose these Boolean values on a column and tests it out. The first `INSERT` command succeeds, while the second one does not. `UPDATE` commands will be subject to this `CHECK` constraint

as well. As mentioned in Chapter 5, user-defined functions are a handy way to implement specialized domain constraints by embedding them in CHECK constraints.

Creating User-Defined Aggregates

Aggregate functions are functions that are applied to all records in a result set and compute some kind of aggregate value from them. SUM(), COUNT(), and AVG() are examples of standard SQL aggregate functions in SQLite.

Implementing user-defined aggregates is a three-step process in which you register the aggregate, implement a step function to be called for each record in the result set, and implement a finalize function to be called after record processing. The finalize function allows you to compute the final aggregate value and do any necessary cleanup.

The following is an example of implementing a simple SUM() aggregate called pysum in one of the SQLite Python extensions:

```
connection=apsw.Connection("foods.db")

def step(context, *args):
    context['value'] += args[0]

def finalize(context):
    return context['value']

def pysum():
    return ({'value' : 0}, step, finalize)

connection.createaggregatefunction("pysum", pysum)

c = connection.cursor()
print c.execute("select pysum(id) from foods").next()[0]
```

The createaggregatefunction() function registers the aggregate, passing in the step function and the finalize function. SQLite passes step() a context, which it uses to store the intermediate value between calls to step(). In this case, it is the running sum. SQLite calls finalize() after it has processed the last record. Here, finalize() just returns the aggregated sum. SQLite automatically takes care of cleaning up the context.

Creating User-Defined Collations

Collations define how string values are compared. User-defined collations therefore provide a way to create different text comparison and sorting methods. This is done in the API by the sqlite3_create_collation() function. SQLite provides three default collations: BINARY, NOCASE, and REVERSE. BINARY compares string values using the C function memcmp() (which for all intents and purposes is case sensitive). NOCASE is just the opposite—its sorting is case insensitive. REVERSE is just an arbitrary method used for testing, and does the direct opposite of BINARY.

User-defined collations are especially helpful for locales that are not well served by the default BINARY collation, or those that need support for UTF-16. They can also be helpful in specific applications such as sorting date formats that don't lend themselves to both lexico-

graphical and chronological order. Chapter 7 illustrates implementing a user-defined collation to sort Oracle dates natively in SQLite.

Transactions

By now you should have a good picture of how the API is laid out. You've seen different ways to execute SQL commands along with some helpful utility functions. Executing SQL commands, however, involves more than just knowing what's in the API. Transactions and locks are closely intertwined with query processing. Queries are always performed within transactions, transactions involve locks, and locks can cause problems if you don't watch what you are doing. You can control both the type and duration of locks by how you use SQL and the way you write code.

Chapter 4 illustrated a specific scenario where deadlocks can arise just by the way that two connections manage transactions through SQL alone. As a programmer, you will have another variable to juggle—code—which can contain multiple connections in multiple states with multiple statement handles on multiple tables at any given time. All it takes is a single statement handle and your code may be holding an `EXCLUSIVE` lock without you even realizing it, preventing other connections from getting anything done.

That is why it is critical that you have good grasp of how both transactions and locks work, and how they relate to the various API functions used to perform queries. Ideally, you should be able to look at the code you write and tell what transaction states it will be in, or at least be able to spot potential problems. In this section we will explore the mechanics behind transactions and locks, and in the next section observe them at work in actual code.

Transaction Lifecycles

There are a couple of things to consider with code and transactions. First there is the issue of knowing which objects run under which transactions. Next there is the question of duration—when does a transaction start and when does it end, and at what point does it start to affect other connections? The first question relates directly to the API. The second relates to SQL in general and SQLite's implementation in particular.

As you know, multiple statement handles can be allocated from a single connection. As shown in Figure 5-2, each connection has exactly one B-tree and one pager object associated with it per database. The pager plays a bigger role than the connection in this discussion because it manages transactions, locks, the memory cache, and crash recovery—all of which will be covered in the next few sections. You could just as easily say that the connection object handles all of this, but in reality it is the pager within it, and that is the object I will oftentimes refer to. The important thing to remember is that when you write to the database, you do so with one connection, one transaction at a time. Thus all statement objects run within the single transaction context of the connections they are derived from. This answers the first question.

As for the second question, transaction duration (or the transaction lifecycle) is as short as a single statement, or as long as you like—until you say stop. By default, a connection operates in autocommit mode, which means that every command you issue runs under a separate transaction. Conversely, when you issue a `BEGIN`, the transaction endures until you call either a COMMIT or a `ROLLBACK`, or until one of your SQL commands causes a constraint violation that results in a `ROLLBACK`. The next question is how transactions relate to locks.

Lock States

For the most part, lock duration shadows transaction duration. While the two don't always start together, they do always finish together. When you conclude a transaction, you free its associated lock. A better way of saying this is that a lock is never released until its associated transaction concludes or, in the worse case, the program crashes. And if the program or system crashes, the transaction doesn't conclude in which case there is still an implicit lock on the database that will be resolved by the next connection to access it. This is covered later in the sidebar "Locking and Crash Recovery."

There are five different locks states in SQLite, and a connection is always in one of them no matter what it's doing. SQLite's lock states and transitions are shown in Figure 5-3. This diagram details every possible lock state a connection can be in as well as every path it can take through the life of a transaction. The diagram is represented in terms of lock states, lock transitions, and transaction lifecycles. What you are really looking at in the figure are the lives of transactions in terms of locks.

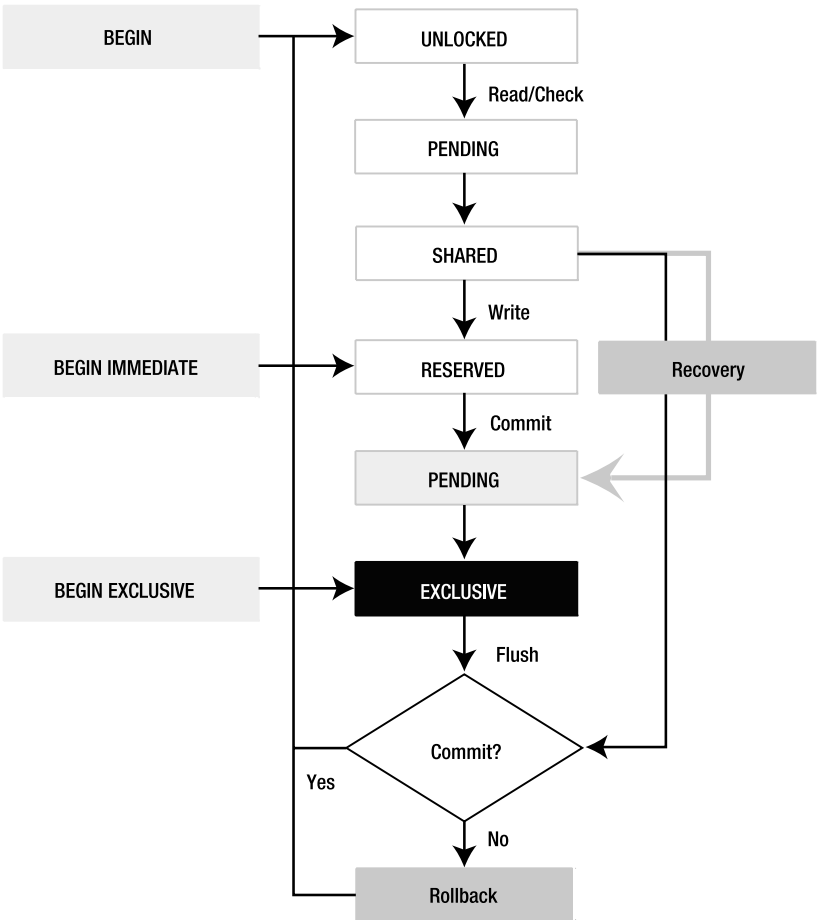


Figure 5-3. *SQLite lock transitions*

Each state has a corresponding lock with the exception of UNLOCKED. So you can say a connection "has a RESERVED lock," or "is in the RESERVED state," or just "is in RESERVED," and it all means the same thing. With the exception of UNLOCKED, for a connection to be in a given state it must first obtain the associated lock.

Every transaction starts at UNLOCKED, RESERVED, or EXCLUSIVE. By default, everything begins in UNLOCKED, as you can see in Figure 5-3. The locks states in white—UNLOCKED, PENDING, SHARED, and RESERVED—can all exist at the same time between connections in a database. Starting with PENDING in gray, however, things become more restrictive. The gray PENDING state represents the lock being held by a single connection—namely a writer that wants to get to EXCLUSIVE. Conversely, the white PENDING state represents a path where connections acquire and release the lock on their way to SHARED. Despite all of these different lock states, every transaction in SQLite boils down to one of two types: transactions that read, and transactions that write. That ultimately is what paints the locking picture: readers versus writers, and how they get along with each other.

Read Transactions

To start with, let's go through the lock process of a SELECT statement. Its path is very simple. A connection that executes a SELECT statement starts a transaction, which goes from UNLOCKED to SHARED and upon COMMIT back to UNLOCKED. End of story.

Now, out of curiosity, what happens when you execute two statements? What is the lock path then? Well, it depends on whether you are running in autocommit or not. Consider the following example:

```
db = open('foods.db')
db.exec('BEGIN')
db.exec('SELECT * FROM episodes')
db.exec('SELECT * FROM episodes')
db.exec('COMMIT')
db.close()
```

Here, with an explicit BEGIN, the two SELECT commands execute within a single transaction and therefore are executed in the same SHARED state. The first exec() runs, leaving the connection in SHARED, and then the second exec() runs. Finally, the manual COMMIT takes the connection from SHARED back to UNLOCKED. The code's lock path would be as follows:

```
UNLOCKED→PENDING→SHARED→UNLOCKED
```

Now consider the case where there are no BEGIN and COMMIT lines in the example. Then the two SELECT commands run in autocommit mode. They will therefore go through the entire path independently. The lock path for the code now would be as follows:

```
UNLOCKED→PENDING→SHARED→UNLOCKED→PENDING→ SHARED→UNLOCKED
```

Since the code is just reading data, it may not make much of a difference, but it does have to go through twice the file locks in autocommit mode than it does otherwise. And, as you will see, a writer could sneak in between the two SELECT commands and modify the database between exec() calls, so you can't be sure that the two commands will return the same results. With BEGIN..COMMIT, on the other hand, they are guaranteed to be identical in their results.

Write Transactions

Now let's consider a statement that writes to the database, such as an UPDATE. First, the connection has to follow the same path as SELECT and get to SHARED. Every operation—read or write—has to start by going through UNLOCKED→PENDING→SHARED. PENDING, as you will soon see, is a gateway lock.

The Reserved State

The moment the connection tries to write anything to the database, it has to go from SHARED to RESERVED. If it gets the RESERVED lock, then it is ready to start making modifications. Even though the connection cannot actually modify the database at this point, it can store modifications in a localized memory cache inside the pager, called the *page cache*, mentioned earlier. This cache is the same cache you configure with the cache_size pragma, as described in Chapter 4.

When the connection enters RESERVED, the pager initializes the *rollback journal*. This is a file (shown in Figure 5-1) that is used in rollbacks and crash recovery. Specifically, it holds the database pages needed to restore the database to its original state before the transaction. These database pages are put there by the pager when the B-tree modifies a page. In this example, for every record the UPDATE command modifies, the pager takes the database page associated with the *original* record and copies it out to the journal. The journal then holds some of the contents of the database before the transaction. Therefore, all the pager has to do in order to undo any transaction is to simply copy the contents in the journal back into the database file. Then the database is restored to its state before the transaction.

In the RESERVED state there are actually three sets of pages that the pager manages: modified pages, unmodified pages, and journal pages. Modified pages are just that—pages containing records that the B-tree has changed. These are stored in the page cache. Unmodified pages are pages the B-tree read but did not change. These are a product of commands such as SELECT. Finally, there are journal pages, which are the original versions of modified pages. These are not stored in the page cache, but rather written to the journal before the B-tree modifies a page.

Because of the page cache, a writing connection can indeed get real work done in the RESERVED state without interfering with other (reading) connections. Thus, SQLite can effectively have multiple readers and one writer both working on the same database at the same time. The only catch is that the writing connection has to store its modifications in its page cache, not in the database file. Note also that there can only be one connection in RESERVED or EXCLUSIVE for a given database at a given time—multiple readers, but only one writer.

The Pending State

When the connection finishes making changes for the UPDATE, and the time comes to commit the transaction, the pager begins the process of entering the EXCLUSIVE state. You already know how this works from Chapter 4, but I will repeat it for the sake of completeness. From the RESERVED state, the pager tries to get a PENDING lock. Once it does, it holds onto it, preventing any other connections from getting a PENDING lock. Look at Figure 5-3 and see what effect this has. Remember I told you that PENDING was a gateway lock. Now you see why. Since the writer is holding onto the PENDING lock, nobody else can get to SHARED from UNLOCKED anymore. The result is that no new connections can enter the database: no new readers, no new writers. This PENDING state is the attrition phase. The writer is guaranteed that it can wait in line for the database and—as long as everyone behaves properly—get it. Only other sessions that already have

SHARED locks can continue to work as normal. In PENDING, the writer waits for these connections to finish and release their locks. What's involved with waiting for locks is a separate issue, which will be addressed shortly in the section "Waiting for Locks."

When the other connections release their locks, the database then belongs to the writer. Then, the pager moves from PENDING to EXCLUSIVE.

The Exclusive State

During EXCLUSIVE, the main job is to flush the modified pages from the page cache to the database file. This is when things get serious, as the pager is going to actually modify the database. It goes about this with extreme caution.

Before the pager begins writing the modified pages, it first tends to the journal. It checks that the complete contents of the journal have been written to disk. At this point, it is very likely that even though the pager has written pages to the journal file, the operating system has buffered many if not all of them in memory. The pager tells the operating system to literally write all of these pages to the disk. This is where the synchronous pragma comes into play, as described in Chapter 4. The method specified by synchronous determines how careful the pager is to ensure that the operating system commits journal pages to disk. The normal setting is to perform a single "sync" before continuing, telling the operating system to confirm that all buffered journal pages are written to the disk surface. If synchronous is set to FULL, then the pager does two full "syncs" before proceeding. If synchronous is set to NONE, the pager doesn't bother with the journal at all (and while it can be 50 times faster, you can kiss transaction durability goodbye).

The reason that committing the journal to disk is so important is that if the program or system crashes while the pager is writing to the database file, the journal is the only way to restore the database file later on. If the journal's pages weren't completely written to disk before a system crash, then the database cannot be restored fully to its original state, because the journal pages that were in memory were lost in the crash. In this case, you have an inconsistent database at best, and a corrupted one at worse.

■ **Caution** Even if you use the most conservative setting for the synchronous pragma, you still may not be guaranteed that the journal is truly committed to disk. This is no fault of SQLite, but rather of certain types of hardware and operating systems. SQLite uses the fsync() system call on Unix and FlushFileBuffers() on Windows to force journal pages to disk. But it has been reported that these functions don't always work, especially with cheap IDE disks. Apparently, some manufacturers of IDE disks use controller chips that tend to bend the truth about actually committing data to disk. In some cases, the chips cache the data in volatile memory while reporting that they wrote it to the drive surface. Also, there have been (unconfirmed) reports that Windows occasionally ignores FlushFileBuffers(). If you have hardware or software that lies to you, your transactions may not be as durable as you might think.

Once the journal is taken care of, the pager then copies all of the modified pages to the database file. What happens next depends on the transaction mode. If, as in this case, the transaction autocommits, then the pager cleans up the journal, clears the page cache, and proceeds from EXCLUSIVE to UNLOCKED. If the transaction does not commit, then the pager

continues to hold the EXCLUSIVE lock and the journal stays in play until either a COMMIT or ROLLBACK is issued.

Autocommit and Efficiency

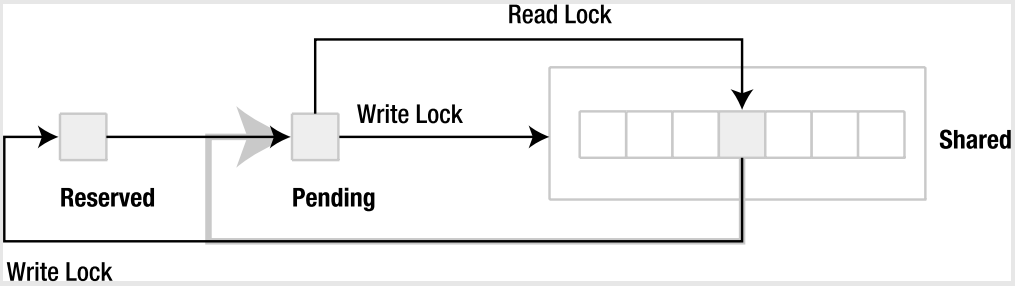
With all this in mind, consider what happens with UPDATEs that run in an explicit transaction versus ones that run in autocommit. In autocommit, every command that modifies the database runs in a separate transaction and travels through the following path:

UNLOCKED→PENDING→SHARED→RESERVED→PENDING→ EXCLUSIVE→UNLOCKED

Additionally, each trip through this path involves creating, committing, and clearing out the rollback journal. This all adds up. Although running multiple SELECTs in autocommit mode is perhaps not that big of a deal performance-wise, you should really rethink autocommit mode for frequent writes. And, as mentioned in the SELECT case, when running multiple commands in autocommit, there is nothing stopping another connection from coming along and modifying the database in between commands. If you have two updates that depend on specific values in the database, you should always run them in the same transaction for this reason.

LOCKING AND CRASH RECOVERY

SQLite's lock implementation is based on standard file locking. SQLite keeps three different file locks on the database file: a reserved byte, a pending byte, and a shared region.



Everything starts at the pending byte. To move from UNLOCKED to SHARED, a connection first attempts to get a read-lock on the pending byte. If successful, it gets a read lock on a random byte in the shared region and releases the read lock on the pending byte. To move from SHARED to RESERVED, a connection attempts to obtain a write lock on the reserved byte. To get from RESERVED to EXCLUSIVE, a connection attempts to get a write lock on the pending byte. If successful, this is what causes the attrition process, as it is no longer possible for other connections to get a read lock on the pending byte to enter SHARED. Finally, to get the EXCLUSIVE lock, the connection attempts to get a write lock on the *entire shared region*. Since the shared region holds read locks by all other active connections, this step guarantees that an EXCLUSIVE lock is only granted after all other SHARED locks are first released.

SQLite's crash recovery mechanism uses the reserved byte to determine when a database needs to be restored. Since the journal file and the RESERVED lock go hand in hand, if the pager sees the former without the latter, then something is wrong. Every time the pager opens a database file or tries to fetch a page from the

database, it does a simple consistency check. If it finds a journal file but no RESERVED lock on the database, then the process that created the journal file must have crashed, or the system went down. In this case, the journal is called a *hot journal*, and the database is potentially in an inconsistent state. To make things right, the journal needs to be "played back" to restore the database to its original state before the interrupted transaction.

To start the play back, the pager puts the database into recovery mode. To do this, it goes directly from SHARED to PENDING as shown by the gray line in the illustration (and also in Figure 5-3). This is the only time it ever makes this transition. The reason it skips the reserved lock is twofold. First, by locking the pending byte, it keeps all new connections out of the database. Second, connections that are already in the database (in SHARED) will also see the hot journal the next time they try to access a page. Those connections will attempt to go into recovery mode and replay the journal as well. However, they won't be able to because the first connection already has the PENDING lock. Thus, by going straight from SHARED to PENDING, the first connection ensures that (1) no new connections can enter the database and (2) active connections in SHARED cannot go into recovery mode. Everyone but the restoring connection is temporarily suspended.

Basically, a hot journal is an implicit EXCLUSIVE lock. If a writer crashes, no further activity can transpire in the database until some connection restores it. The next pager to access a page will see the hot journal, lock everyone out, and start recovery. If there are no other active connections, then the first program to connect to the database will detect the hot journal and start recovery.

Tuning the Page Cache

Go back to the beginning of the previous example and say that it started with a BEGIN, followed by the UPDATE, and in the middle of making all those modifications the page cache fills up (runs out of memory). That is, the UPDATE results in more modified pages than will fit in the page cache. What happens now?

Transitioning to Exclusive

The real question is: when exactly does the pager move from RESERVED to EXCLUSIVE and why? There are two scenarios, and you've just seen them both. Either the connection reaches the commit point and deliberately enters EXCLUSIVE, or the page cache fills up and it has no other option. We just looked at the commit point scenario. So what happens when the page cache fills up? Put simply, the pager can no longer store any more modified pages, and therefore can no longer do any work. It is forced to move into EXCLUSIVE in order to continue. In reality, this is not entirely true as there is a soft limit and a hard limit.

The soft limit corresponds to the first time the page cache fills up. At this point, the cache is a mixed bag of modified and unmodified pages. In this case, the pager tries to clean out the page cache. It goes through the cache page by page looking for unmodified pages and clearing them out. Once it does that, it can sputter along with what memory has freed up until the cache fills up again. It repeats the process until the cache is completely made up of modified pages. And that is that hard limit. At this point, the pager has no other recourse but to proceed into EXCLUSIVE.

The RESERVED state, then, is where the cache_size pragma makes a difference. Just as explained in Chapter 4, cache_size controls the size of the page cache. The bigger the page cache, the more modified pages the pager can store, and the more work the connection can do before having to enter EXCLUSIVE. Also, as mentioned, by doing all the database work in RESERVED,

you minimize the time in EXCLUSIVE. If you get all your work done in RESERVED, then EXCLUSIVE is only held long enough to flush modified pages to disk—not compile *more* queries and process *more* results and *then* write to disk. Doing the processing in RESERVED can significantly increase overall concurrency. Ideally, if you have a large transaction or a congested database, and you can spare the memory, try to ensure that your cache size is large enough to hold your connection in RESERVED as long as possible.

Sizing the Page Cache

So how do you determine what the cache size should be? It depends on what you are doing. Say that you want to update every record in the episodes table. In this case, you know that every page in the table will be modified. Therefore, you figure out how many pages are in episodes and adjust the cache size accordingly. You can get all the information you need on episodes using sqlite_analyzer. For each table it will dump detailed statistics, including the total page count. For example, if you run it on the foods database, you get the following information about episodes:

```
*** Table EPISODES ************************************************

Percentage of total database.....  20.0%
Number of entries..................... 181
Bytes of storage consumed............. 5120
Bytes of payload...................... 3229        63.1%
Average payload per entry............. 17.84
Average unused bytes per entry........ 5.79
Average fanout........................ 4.00
Maximum payload per entry............. 38
Entries that use overflow............. 0            0.0%
Index pages used...................... 1
Primary pages used.................... 4
Overflow pages used................... 0
Total pages used...................... 5
Unused bytes on index pages........... 990         96.7%
Unused bytes on primary pages......... 58           1.4%
Unused bytes on overflow pages........ 0
Unused bytes on all pages............. 1048        20.5%
```

The total page count is 5. But of those, only 4 pages of actual table are used—1 page is an index. Since the default cache size is 2,000 pages, you've got nothing to worry about. There are about 400 records in episodes, which means there are about 100 records per page. You wouldn't have to worry about adjusting the page cache before updating every record unless there were at least 196,000 rows in episodes. And remember, you would only need to do this in environments where there are other connections using the database and concurrency is an issue. If you are the only one using the database, then it really wouldn't matter.

Waiting for Locks

We talked earlier about the pager waiting to go from PENDING to EXCLUSIVE. What exactly is involved with waiting on a lock? First, any call to exec() or step() can involve waiting on a lock. Whenever SQLite encounters a situation where it can't get a lock, the default behavior is to return Delete SQLITE_BUSY to the function that caused it to seek the lock. Regardless of the command you execute, you can potentially encounter SQLITE_BUSY. SELECT commands, as you know by now, can fail to get a SHARED lock if a writer is pending or writing. The simple thing to do when you get SQLITE_BUSY is to just retry the call. However, we will see shortly that this is not always the best course of action.

Using a Busy Handler

Instead of just retrying the call over and over, you can use a *busy handler*. Rather than having the API return SQLITE_BUSY if a connection cannot get a lock, you can get it to call the busy handler instead.

A busy handler is a function you create that kills time, or does whatever else you want it to do—it can send spam to your mother-in-law for all SQLite cares. It's just going to get called when SQLite can't get a lock. The only thing the busy handler *has* to do is provide a return value, telling SQLite what to do next. By convention, if the handler returns true, then SQLite will continue to try for the lock. If it returns false, SQLite will then return SQLITE_BUSY to the function requesting the lock. Consider the following example:

```
counter = 1

def busy()
    counter = counter + 1
    if counter == 2
        return 0
    end

    spam_mother_in_law(100)
    return 1
end

db.busy_handler(busy)
stmt = db.prepare('SELECT * FROM episodes;')
stmt.step()
stmt.finalize()
```

The implementation of spam_mother_in_law() is left as an exercise for the reader.

The step() function has to get a SHARED lock on the database to perform the SELECT. However, say there is a writer active. Normally, step() would return SQLITE_BUSY. However, in this case it doesn't. The pager (which is one that deals with locks) calls the busy() function instead, because it has been registered as the busy handler. busy() increments a counter, forwards your mother-in-law 100 random messages from your spam folder, and returns 1, which the pager

interprets as true—keep trying to get the lock. The pager then tries again to get the SHARED lock. Say the database is still locked. The pager calls the busy handler again. Only this time, busy() returns 0, which the pager interprets as false. In this case, rather than retrying the lock, the pager sends SQLITE_BUSY up the stack and that's what step() ends up returning.

If you just want to kill time waiting for a lock, you don't have to write your own busy handler. The SQLite API has one for you. It is a simple busy handler that sleeps for a given period of time waiting for a lock. In the API, the function is called sqlite3_busy_timeout(), and it is supported by some extension libraries. You can essentially say "try sleeping for 10 seconds when you can't get a lock," and the pager will do that for you. If it sleeps for 10 seconds and still can't get the lock, then it will return SQLITE_BUSY.

Using the Right Transaction

Let's consider the previous example again, but this time the command is an UPDATE rather than a SELECT. What does SQLITE_BUSY actually mean now? In SELECT, it just means, "I can't get a SHARED lock." But what does is mean for an UPDATE? The truth is, you don't really know what it means. SQLITE_BUSY could mean that the connection failed to get a SHARED lock because there is a writer pending. It could also mean that it got a SHARED lock but couldn't get to RESERVED. The point is you don't know the state of the database, or the state of your connection for that matter. In autocommit mode, SQLITE_BUSY for a write operation is completely indeterminate. So what do you do next? Should you just keep calling step() over and over until the command goes through?

Here's the thing to think about. Suppose SQLITE_BUSY was the result of you getting a SHARED lock but not RESERVED, and now you are holding up a connection in RESERVED from getting to EXCLUSIVE. Again, *you don't know the state of the database.* And just using a brute-force method to push your transaction through is not necessarily going to work, for you or any other connection. If you just keep calling step(), you are just going to butt heads with the connection that has the RESERVED lock, and if neither of you backs down, you'll deadlock.

■ **Note** SQLite tries to help with deadlock prevention in this particular scenario by ignoring the busy handler of the offending connection. The SHARED connection's busy handler will not be invoked if it is preventing a RESERVED connection from proceeding. However, it is up to the code to get the hint. The code can still just repeatedly call step() over and over, in which case there is nothing more SQLite can do to help.

Since you know you want to write to the database, then you need to start by issuing BEGIN IMMEDIATE. If you get a SQLITE_BUSY, then at least you know what state you're in. You know you can safely keep trying without holding up another connection. And once you finally do succeed, you know what state you are in then as well—RESERVED. Now you can use brute force if you have to because you are the one in the right. If you start with a BEGIN EXCLUSIVE, on the other hand, then you are assured that you won't have any busy conditions to deal with at all. Just remember that in this case you are doing your work in EXCLUSIVE, which is not as good for concurrency as doing the work in RESERVED.

LOCKS AND NETWORK FILE SYSTEMS

At this point, you should have a good appreciation for what can go wrong when a database is shared over a network file system. SQLite handles concurrency by placing file locks on the database file. It is very important that these locks be both set and released at the right times. SQLite is completely dependent on the file system to manage locks correctly for concurrent use. SQLite uses the same locking mechanisms regardless of whether it is running on a normal file system or network file system. It uses POSIX advisory locks on Unix and the `LockFile()`, `LockFileEx()`, and `UnlockFile()` system calls on Windows. These calls are standard system calls, and work correctly on normal file systems. It is the network file system's job to emulate a normal file system. And unfortunately, this doesn't always work correctly with some implementations. And even if the network file system works correctly, there still other things to consider.

Take NFS, for example. It's a great network file system. However, many original NFS implementations were known to have buggy or in some cases unimplemented locking. And with a SQLite database, this can cause serious problems. Without locking, two connections can get an `EXCLUSIVE` lock on the same database and write to it at the same time, leading to an almost certain database corruption. Or perhaps locking is implemented and a bug does not release the reserved byte or pending byte in a timely manner, or at all. The problems range from delays to database corruption. This is not a problem with the NFS protocol in general, but with some implementations of it in particular. Eric Kustarz, a Sun engineer who has worked on NFS for five years, says the following on his blog regarding NFS locking:

> *The protocol behind locking (NLM) for NFSv2/v3 is not broken*, but rather some of the implementations were broken—especially early on (people spent time on making the NFS protocol work, but NLM was more complex—an after thought)… Running a Solaris client against a Solaris server will find that NFSv3 + NLM works great, and is not broken. Thankfully, NFSv4 fixed this by integrating locking into the one and only protocol. Now, locking has to work or you don't have a NFSv4 implementation.*

Now consider a footnote on the same page:

> **OK, there are a couple very edge cases in NLM that got sorted out in NFSv4. The easiest example is recovery of a client crash…if the client never comes back up, and another client needs to lock the same file? You need administrative intervention at this point. Since NFSv4's state is lease based, no intervention is necessary—if the client doesn't reclaim its state within the allotted time, the locks are lost.*

NFS can still have issues with locking if one of the clients goes down (taking their database lock with it) and does not come back up. For NFSv3, a locked file would need administrative intervention to clear it. For NFSv4, there would be a timeout period. But how long is the timeout? Thirty seconds? A minute? The real issue is not NFS, but specific applications of it. Network file systems, even if perfectly implemented, are not necessarily going to work exactly like a local file system.

The bottom line is, if you are going to mix concurrency and network file systems, there are many issues to consider, even if you are using the best network file system out there. It's not as simple as running on a local file system. There are new variables introduced when any application runs over a network. The real issue is not that SQLite doesn't work on network file systems. It can. Nor is the real issue that network file systems don't work reliably. Any valid implementation of NFS does. The issue is what your application does, what environment it is running in, what network file system(s) it is using (e.g., Linux client and Solaris server), how many other connections are hitting the database, what the specific performance requirements are... the list goes on.

No one can definitively say, "Yes, this network file system will work for you," or "No, SQLite doesn't work." These are only two of many possible variables in the equation—an equation that only you can solve based on your resources and requirements.

Code

By now, you have a pretty good picture of the API, transactions, and locks. To finish up, let's put all three of these things together in the context of your code, and consider a few scenarios that you might want to watch for.

Using Multiple Connections

If you have written code that uses other relational databases, you may have made some assumptions that might have worked with those databases that will not work with SQLite. Many times I have used multiple connections with other databases that work concurrently in a single block of code. The classic example is having one connection iterate over a table while another connection modifies records in place.

In SQLite, using multiple connections in the same block of code can cause problems. You have to be careful of how you go about it. Consider the following example:

```
c1 = open('foods.db')
c2 = open('foods.db')

stmt = c1.prepare('SELECT * FROM episodes')

while stmt.step()
  print stmt.column('name')
  c2.exec('UPDATE episodes SET …)
end

stmt.finalize()

c1.close()
c2.close()
```

I'll bet you can easily spot the problem here. In the `while` loop, `c2` attempts an `UPDATE` while `c1` has a `SHARED` lock open. That `SHARED` lock won't be released until `stmt` is finalized after the `while` loop. Therefore, it is impossible to write to the database within the `while` loop. Either the updates will silently fail, or if you have a busy handler then it will only delay the program. The best thing to do here is to use one connection for the job, and to run it under a single `BEGIN IMMEDIATE` transaction. The new version might be

```
c1 = open('foods.db')

# Keep trying until we get it
while c1.exec('BEGIN IMMEDIATE') != SQLITE_SUCCESS
end

stmt = c1.prepare('SELECT * FROM episodes')

while stmt.step()
  print stmt.column('name')
  c1.exec('UPDATE episodes SET …)
end

stmt.finalize()
c1.exec('COMMIT')
c1.close()
```

In cases like this, you should use statements from a single connection for reading or writing. Then you won't have to worry about database locks causing problems. However, as it turns out, this particular example still won't work. If you are iterating over a table with one statement and updating it with another, there is an additional locking issue that you need to know about as well, which we'll cover next.

Table Locks

Even if you are using just one connection, there is a special edge case that sometimes trips people up. While you would think that two statements from the same connection could work on the database with impunity, there is one important exception.

When you execute a `SELECT` command on a table, the resulting statement object creates a B-tree cursor on that table. As long as there is a B-tree cursor active on a table, other statements—even in the same connection—cannot modify it. If they try, they will get `SQLITE_BUSY`. Consider the following example:

```
c = sqlite.open("foods.db")

stmt1 = c.compile('SELECT * FROM episodes LIMIT 10')

while stmt1.step() do
    # Try to update the row
    row = stm1.row()
    stmt2 = c.compile('UPDATE episodes SET …')
    # Uh oh: ain't gonna happen
    stmt2.step()
end

stmt1.finalize()
stmt2.finalize ()

c.close()
```

We are only using one connection here. Regardless, when `stmt2.step()` is called, it won't work because `stmt1` has a cursor on the `episodes` table. In this case, `stmt2.step()` may actually succeed in promoting the connection's database lock to `EXCLUSIVE`, but it will still return `SQLITE_BUSY`. The cursor on `episodes` prevents it from modifying the table. In order to get around this, you can do one of two things:

- Iterate over the results with one statement, storing the information you need in memory. Then finalize the reading statement, and then do the updates.
- Store the `SELECT` results in a temporary table (as described in a moment) and open the read cursor on it. In this case you can have both a reading statement and a writing statement working at the same time. The reading statement's cursor will be on a different table—the temporary table—and won't block the updates on the main table from the second statement. Then when you are done, simply drop the temporary table.

When a statement is open on a table, its B-tree cursor will be removed from the table when one of two things happens:

- The statement reaches the end of the result set. When this happens, `step()` will automatically close the statement's cursor(s). In VDBE terms, when the end of the results set is reached, the VDBE encounters a `Close` instruction, which causes all associated cursors to be closed.
- The statement is finalized. The program explicitly calls `finalize()`, thereby removing all associated cursors.

In many extensions, the call to `sqlite3_finalize()` is done automatically in the statement object's `close()` function, or something similar.

■ **Note** As a matter of interest, there are exceptions to these scenarios where you could theoretically get away with reading and writing to the same table at the same time. In order to do so, you would have to convince the optimizer to use a temporary table, using something like an `ORDER BY`, for example. When this happens, the optimizer will automatically create a temporary table for the `SELECT` statement and place the reading statement's cursor on it rather than the actual table itself. In this case, it is technically possible for a writer to then modify the real table because the reader's cursor is on a temporary table. The problem with this approach is that the decision to use temporary tables is made by the optimizer. It is not safe to presume what the optimizer will and will not do. Unless you like to gamble, or are just intimately acquainted with the ins and outs of the optimizer, it is best to just follow the general rule of thumb: don't read and write to the same table at the same time.

Fun with Temporary Tables

Temporary tables let you bend the rules. If you absolutely have to have two connections going in the same block of code, or two statements operating on the same table, you can safely do so if you use temporary tables. When a connection creates or writes to a temporary table, it does not have to get a `RESERVED` lock, because temporary tables are maintained outside of the database

file. There are two ways of going about this, depending on how you want to manage concurrency. Consider the following example:

```
c1 = open('foods.db')
c2 = open('foods.db')

c2.exec('CREATE TEMPORARY TABLE temp_epsidodes as SELECT * from episodes')

stmt = c1.prepare('SELECT * FROM episodes')

while stmt.step()
  print stmt.column('name')
  c2.exec('UPDATE temp_episodes SET …')
end

stmt.finalize()

c2.exec('BEGIN IMMEDIATE')
c2.exec('DELETE FROM episodes')
c2.exec('INSERT INTO episodes SELECT * FROM temp_episodes')
c2.exec('COMMIT')

c1.close()
c2.close()
```

This example uses a temporary table based on `episodes` to make modifications within the `while` loop. Both connections are actually running in `SHARED`. Again, this is because temporary tables are maintained outside of the database. Therefore, `temp_episodes` and all of the operations on it don't require that `c2` gets a `RESERVED` or an EXCLUSIVE lock. Once `stmt` is finalized, then `c1`'s `SHARED` lock is gone and `c2` can safely write to the database all that it collected in the temporary table.

Notice also that this example uses the original `episodes` table rather than dropping it and re-creating it from `temp_episodes` using a `CREATE TABLE AS SELECT…`. If you did it this way, then you would lose any integrity constraints and indexes defined on the original `episodes` table.

The advantage of the previous example is that it minimizes the duration of `RESERVED` and `EXCLUSIVE` locks on the database. Everything is kept in a temporary table during the iteration, so it does not affect other reading connections. Then everything is copied over in one single transaction. The code could be simplified somewhat, but at the cost of decreased concurrency as follows:

```
c1 = open('foods.db')
c2 = open('foods.db')

c1.exec('CREATE TEMPORARY TABLE temp_episodes as SELECT * from episodes')

stmt = c1.prepare('SELECT * FROM temp_episodes')
```

```
while stmt.step()
  print stmt.column('name')
  c2.exec('UPDATE episodes SET …') # What about SQLITE_BUSY?
end

stmt.finalize()

c1.exec('DROP TABLE temp_episodes')

c1.close()
c2.close()
```

This example stores the result set into the temporary table, iterates over it, and performs the updates to the main table. While this code is a little cleaner, keep in mind that you now have additional concurrency issues to deal with. First, there will now be `RESERVED` and `EXCLUSIVE` locks associated with each `UPDATE` where there weren't any before. Second, you have to guard the updates against `SQLITE_BUSY` conditions (not shown in the example). Finally, if you run the whole thing under a single transaction, it may have an `EXCLUSIVE` lock on the database during the entire iteration, depending on how much is updated and the size of the page cache. All in all, the first example, while involving a little more code, may be the better approach in general.

The Importance of Finalizing

A common gotcha in processing `SELECT` statements is the failure to realize that the `SHARED` lock is not released until `finalize()` is called—well, most of the time. Consider the following example:

```
stmt = c1.prepare('SELECT * FROM episodes')

while stmt.step()
  print stmt.column('name')
end

c2.exec('BEGIN IMMEDIATE; UPDATE episodes SET …; COMMIT;')

stmt.finalize()
```

While you should never do this in practice, you might end up doing it anyway by accident simply because you can get away with it. If you write the equivalent of this program in the C API, it will actually work. Even though `finalize()` has not been called, the second connection can modify the database without any problem. Before I tell you why, consider this next example:

```
c1 = open('foods.db')
c2 = open('foods.db')

stmt = c1.prepare('SELECT * FROM episodes')
```

```
stmt.step()
stmt.step()
stmt.step()

c2.exec('BEGIN IMMEDIATE; UPDATE episodes SET …; COMMIT;')

stmt.finalize()
```

Let's say that `episodes` has 100 records. And the program only stepped through three of them. What happens here? The second connection will get `SQLITE_BUSY`.

In the first example, SQLite released the `SHARED` lock when the statement reached the end of the result set. That is, in the final call to `step()`, where the API returns `SQLITE_DONE`, the VDBE encountered the `Close` instruction and SQLite closed the cursor and dropped the `SHARED` lock. Thus, `c2` was able to push its `INSERT` through even though `c1` had not called `finalize()`.

In the second case, the statement had not reached the end of the set. The next call to `step()` would have returned `SQLITE_RESULT`, which means there are more rows in the results set, and that the `SHARED` lock is still active. Thus, `c2` could not get the `INSERT` through this time because of the `SHARED` lock from `c1`.

The moral of the story is don't do this, even though sometimes you can. Always call `finalize()` before you write with another connection. The other thing to remember is that in autocommit mode `step()` and `finalize()` are more or less transaction and lock boundaries. They start and end transactions. They acquire and release locks. You should be very careful about what you do with other connections in between these functions.

Shared Cache Mode

Now that you are clear on the concurrency rules, I will give you something new to confuse you. SQLite offers an alternative concurrency model called *shared cache mode*, which relates to how connections can operate within individual threads.

In shared cache mode, a thread can create multiple connections that share the same page cache. Furthermore, this group of connections can have multiple readers *and a single writer* (in `EXCLUSIVE`) working on the same database at the same time. The catch is that these connections cannot be shared across threads—they are strictly limited to the thread (running specifically in shared cache mode) that created them. Furthermore, writers and readers have to be prepared to handle a special condition involving table locks.

When readers read tables, SQLite automatically puts table locks on them. This prevents writers from modifying those tables. If a writer tries to modify a read-locked table, it will get `SQLITE_LOCKED`. The same logic applies to readers trying to read from write-locked tables. However, in this latter case, readers can still go ahead and read tables that are being modified by a writer if they run in *read-uncommitted mode*, which is enabled by the `read_uncommitted` pragma. In this case, SQLite does not place read locks on the tables read by these readers. As a result, these readers don't interfere with writers at all. However, these readers can get inconsistent query results, as a writer can modify tables as the readers read them.

Shared cache mode is designed for embedded servers that need to conserve memory and have slightly higher concurrency under certain conditions. More information on using it with the C API can be found in Chapter 6.

Summary

The SQLite API is flexible, intuitive, and easy to use. It has two basic parts: the core API and the extension API. The core API revolves around two basic data structures used to execute SQL commands: the connection and the statement. Commands are executed in three steps: compilation, execution, and finalization. SQLite's wrapper functions `exec()` and `get_table()` wrap these steps into a single function call, automatically handling the associated statement object for you. Other parts of the core API include support for string handling, operational control, and other miscellaneous utility functions.

The extension API provides you with the means to customize SQLite in three difference ways: user defined functions, user-defined aggregates, and user-defined collations. Both user-defined functions and aggregates can be implemented in different languages and called directly from SQL. User-defined collations allow SQLite to be more usable in different locales.

Because SQLite's concurrency model is somewhat different from other databases, it is important that you understand a bit about how it manages transactions and locks, how they work behind the scenes and within your code. Overall, the concepts are not difficult to understand, and there are just a few simple rules to keep in mind when you write code that uses SQLite.

The next three chapters will draw heavily on what you've have learned here, as these concepts apply not only to the C API, but to language extensions as well as they are built on top of the C API.