CHAPTER 6

The Core C API

This chapter covers the part of the SQLite API used to work with databases. You already have a good idea of how the API works from Chapter 5. Now let's concentrate on the specifics.

Starting with a few trivial examples, we will take an in-depth tour through the C API and expand upon the examples, filling in various details with a variety of useful functions. As we go along, you should see the C equivalents of the model all fall into place, with some additional features you may not have seen before—features that primarily exist only in the C API. By the time we reach the end of this chapter, you should have seen every API function related to running commands, managing transactions, fetching records, handling errors, and many other tasks related to general database work.

The SQLite version 3 API consists of approximately 80 functions. Only about eight functions, however, are needed to actually connect, process queries, and disconnect from a database. The remaining functions can be arranged into small groups that specialize in accomplishing specific tasks.

As mentioned in Chapter 5, quite a few things have changed in the API between versions 2 and 3. One of the most notable is the addition of UTF support. All functions that accept character strings as function arguments, or produce them as return values, have both UTF-8 and UTF-16 analogs. For example, `sqlite3_open()`, which is used to open a database, takes a UTF-8 string containing a database filename as an argument. Its counterpart—`sqlite3_open16()`—has the exact same signature, but accepts the same argument in UTF-16 encoding. With the exception of the first section, I refer only to the UTF-8 functions, as the UTF-16 versions differ only slightly by their names.

While it is best to read the chapter straight through, if at any point you want more detailed information on a particular function, you can consult Appendix B, which contains the complete C API documentation.

All examples in this chapter can be found in self-contained example programs, the source files of which are located in the *ch6* folder of the examples zip file, available on the Apress website at `www.apress.com`. For every example presented, I specifically identify the name of the corresponding source file from which the example was taken.

Wrapped Queries

You are already familiar with the way that SQLite executes queries, as well as its various wrapper functions for executing SQL commands in a single function call. We will start with the C API versions of these wrappers because they are simple, self-contained, and easy to use. They are a

good starting point, which will let you have some fun and not get too bogged down with details. Along the way, I'll introduce some other handy functions that go hand in hand with query processing. By the end of this section, you will be able connect, disconnect, and query a database using the wrapped queries.

Connecting and Disconnecting

Before you can execute SQL commands, you first have to connect to a database. Connecting to a database is perhaps best described as *opening* a database, as SQLite databases are contained in single operating system files (one file to one database). Similarly, the preferred term for disconnecting would be *closing* the database.

You open a database with the `sqlite3_open()` or `sqlite3_open16()` functions, which have the following declaration(s):[1]

```
int sqlite3_open(
  const char *filename,   /* Database filename (UTF-8) */
  sqlite3 **ppDb          /* OUT: SQLite db handle */
);

int sqlite3_open16(
  const void *filename,   /* Database filename (UTF-16) */
  sqlite3 **ppDb          /* OUT: SQLite db handle */
);
```

The `filename` argument can be the name of an operating system file, the text string `':memory:'`, or an empty string (`NULL` pointer). If you use `NULL` or `':memory:'`, `sqlite3_open()` will create an in-memory database in RAM that lasts only for the duration of the session. If `filename` is not `NULL`, `sqlite3_open()` attempts to open the database file by using its value. If no file by that name exists, `sqlite3_open()` will open a new database file by that name.

■ **Note** When SQLite creates a new database, it does not actually write anything to disk until the first database object is created. Therefore, if you specify a new database file but do not create a table, view, trigger, or other database object, when you close the database the database file by that name will still not exist in the file system. Therefore, to actually create a new database, you must first create something in it before you close it. Other important considerations associated with creating new databases are covered in Chapter 5.

Upon completion, `sqlite3_open()` will initialize the `sqlite3` structure passed into it by the `ppDb` argument. This structure should be considered as an opaque handle representing a single connection to a database. This is more of a connection handle than a database handle since it

1. Here I have included both the UTF-8 and UTF-16 declarations for `sqlite3_open()`. From here on out, I will refer to the UTF-8 declarations only for the sake of brevity. Therefore, please keep in mind that there are many functions in the API that have UTF-16 forms. All of these forms are listed in the API documentation in Appendix B.

is possible to attach multiple databases to a single connection. However, this connection still represents exactly one transaction context regardless of how many databases are attached.

If a connection's transaction context is not explicitly defined using `BEGIN`, `COMMIT`, and `ROLLBACK`, SQLite will by default operate in autocommit mode, where every statement issued to the connection is run under a separate transaction.

You close a connection by using the `sqlite3_close()` function, which is declared as follows:

```
int sqlite3_close(sqlite3*);
```

In order for `sqlite3_close()` to complete successfully, all prepared queries associated with the connection must be finalized. If any queries remain that have not been finalized, `sqlite3_close()` will return `SQLITE_BUSY` with the error message *Unable to close due to unfinalized statements*.

■ **Note** If there is a transaction open on a connection when it is closed by `sqlite3_close()`, the transaction will automatically be rolled back.

The exec Query

The `sqlite3_exec()` function provides a quick, easy way to execute SQL commands and is especially handy for commands that modify the database (i.e., don't return any data). This function has the following declaration:

```
int sqlite3_exec(
  sqlite3*,                    /* An open database */
  const char *sql,             /* SQL to be executed */
  sqlite_callback,             /* Callback function */
  void *data                   /* 1st argument to callback function */
  char **errmsg                /* Error msg written here */
);
```

The SQL provided in the `sql` argument can consist of more than one SQL command. `sqlite3_exec()` will parse and execute every command in the `sql` string until it reaches the end of the string or encounters an error. Listing 6-1 (taken from `create.c`) illustrates using `sqlite3_exec()`.

Listing 6-1. *Using sqlite3_exec() for Simple Commands*

```
int main(int argc, char **argv)
{
    sqlite3 *db;
    char *zErr;
    int rc;
    char *sql;
```

```
    rc = sqlite3_open("test.db", &db);

    if(rc) {
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        exit(1);
    }

    sql = "create table episodes(id int, name text)";
    rc = sqlite3_exec(db, sql, NULL, NULL, &zErr);

    if(rc != SQLITE_OK) {
        if (zErr != NULL) {
            fprintf(stderr, "SQL error: %s\n", zErr);
            sqlite3_free(zErr);
        }
    }

    sql = "insert into episodes values (10, 'The Dinner Party')";
    rc = sqlite3_exec(db, sql, NULL, NULL, &zErr);

    sqlite3_close(db);
    return 0;
}
```

This example opens a database `test.db` and creates within it a single table called `episodes`. After that, it inserts one record. The `CREATE TABLE` command will physically create the database file if it does not already exist.

Processing Records

As mentioned in Chapter 5, it is actually possible to get records from `sqlite3_exec()`, although you don't see it implemented much outside of the C API. `sqlite3_exec()` contains a callback mechanism that provides a way to obtain results from `SELECT` statements. This mechanism is implemented by the third and fourth arguments of the function. The third argument is a pointer to a callback function. If it's provided, SQLite will call the function for each record processed in each `SELECT` statement executed within the `sql` argument. The callback function has the following declaration:

```
typedef int (*sqlite3_callback)(
    void*,    /* Data provided in the 4th argument of sqlite3_exec() */
    int,      /* The number of columns in row                         */
    char**,   /* An array of strings representing fields in the row  */
    char**    /* An array of strings representing column names       */
);
```

The fourth argument to `sqlite3_exec()` is a void pointer to any application-specific data you want to supply to the callback function. SQLite will pass this data as the first argument of the callback function.

The final argument (errmsg) is a pointer to a string to which an error message can be written should an error occur during processing. Thus, sqlite3_exec() has two sources of error information. The first is the return value. The other is the human-readable string, assigned to errmsg. If you pass in a NULL for errmsg, then SQLite will not provide any error message. Note that if you do provide a pointer for errmsg, the memory used to create the message is allocated on the heap. You should therefore check for a non-NULL value after the call and use sqlite3_free() to free the memory used to hold the errmsg string if an error occurs.

Putting it all together, sqlite3_exec() allows you to issue a batch of commands, and you can collect all of the returned data by using the callback interface. For example, let's insert a record into the episodes table and then select all of its records, all in a single call to sqlite3_exec(). The complete code, shown in Listing 6-2, is taken from exec.c.

Listing 6-2. *Using sqlite3_exec() for Record Processing*

```
int callback(void* data, int ncols, char** values, char** headers);

int main(int argc, char **argv)
{
    sqlite3 *db;
    int rc;
    char *sql;
    char *zErr;

    rc = sqlite3_open("test.db", &db);

    if(rc) {
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        exit(1);
    }

    const char* data = "Callback function called";
    sql = "insert into episodes (cid, name) values (11,'Mackinaw Peaches');"
          "select * from episodes;";
    rc = sqlite3_exec(db, sql, callback, data, &zErr);

    if(rc != SQLITE_OK) {
        if (zErr != NULL) {
            fprintf(stderr, "SQL error: %s\n", zErr);
            sqlite3_free(zErr);
        }
    }

    sqlite3_close(db);
    return 0;
}
```

```
int callback(void* data, int ncols, char** values, char** headers)
{
    int i;
    fprintf(stderr, "%s: ", (const char*)data);
    for(i=0; i < ncols; i++) {
        fprintf(stderr, "%s=%s ", headers[i], values[i]);
    }

    fprintf(stderr, "\n");
    return 0;
}
```

SQLite parses the `sql` string; runs the first command, which inserts a record; and then runs the second command, consisting of the `SELECT` statement. For the second command, SQLite calls the callback function for each record returned. Running the program produces the following output:

```
Callback function called: id=10 name=The Dinner Party
Callback function called: id=11 name=Mackinaw Peaches
```

Notice that the callback function returns 0. This return value actually exerts some control over `sqlite3_exec()`. If the callback function returns a non-zero value, then `sqlite3_exec()` will abort (in other words, it will terminate all processing of this and subsequent commands in the `sql` string).

So `sqlite3_exec()` provides not only an easy way to modify the database but an interface with which to process records as well. Why then should you bother with prepared queries? Well, as you will see in the next section, there are quite a few advantages to using the latter:

- Prepared queries don't require a callback interface, which makes coding simple and more linear.
- Prepared queries have associated functions that provide better column information. You can obtain a column's storage type, declared type, schema name (if it is aliased), table name, and database name. `sqlite3_exec()`'s callback interface provides just the column names.
- Prepared queries provide a way to obtain field/column values in other data types besides text, in native C data types such as `int` and `double`, whereas `sqlite3_exec()`'s callback interface only provides fields as string values.
- Prepared queries can be rerun, allowing you to reuse the compiled SQL.
- Prepared queries support parameterized SQL statements.

As a matter of history, `sqlite3_exec()`'s callback interface is reminiscent of the old SQLite 2.x API.[2] In that API, the callback interface was the way you performed all queries

2. It is interesting to note that `sqlite3_exec()` is implemented in a source file called `legacy.c`.

and retrieved all records. The new approach is a refinement of this interface, which works more like other database client libraries.

EXAMINING CHANGES

If you are performing an UPDATE or a DELETE, you may want to know how many records were affected. You can get this information from sqlite3_changes(). It provides the number of affected records for the last executed UPDATE, INSERT, or DELETE statement. Obviously, if you are running a batch of queries (multiple commands in the sql argument of sqlite3_exec()), this function will only be good for the last command in the batch. There are a few other caveats as well:

- Auxiliary changes caused by triggers are not counted. In order to obtain affected records from triggers, use the sqlite3_total_changes() function.
- When deleting *all* records in a table (delete from xyz), sqlite3_changes() will return 0 records. The reason for this is that SQLite optimizes this statement by dropping and re-creating the table in question rather than deleting individual records. If you need to know the actual number of deleted records, modify your DELETE statement to the form DELETE FROM table_name WHERE 1. This will disable the drop/re-create optimization and return the actual number of deleted records. Of course, you could also perform a SELECT count()... on the table in question before deleting all records to get the number of deleted records.

If you are performing an INSERT on a table with an autoincrement column, odds are you will eventually want to know the primary key value of an inserted record. In such cases, you can use sqlite3_last_insert_rowid() to obtain this value. You can also do this from within SQL as well via the last_insert_rowid() SQL function.

String Handling

As discussed in Chapter 5, SQLite includes functions for string handling. If your program has to deal with user input or parameters, some extremely handy functions to know about are SQLite's formatting functions, which are declared as follows:

```
char *sqlite3_mprintf(const char* sql, arg1, arg2, ...);
char *sqlite3_vmprintf(const char*, va_list);
```

The first function, sqlite3_mprintf(), works like sprintf(), but has features specific to SQL formatting. The handy argument here is %q, which is an enhanced %s. When used, %q will automatically escape SQL-sensitive characters in the substituted string. Consider the following code:

```
char *sql;
char *trouble = "'Here's trouble'";
sql = sqlite3_mprintf("insert into x values('%q')", trouble);
/* do something */
sqlite3_free(sql);
```

The result sql will contain

```
    insert into x values('''Here''s trouble''')
```

The %Q format does everything %q does, but it also encloses the resulting string in single quotes. Furthermore, if the argument for the %Q formatting options is a NULL pointer, sqlite3_mprintf() will produce the string NULL without single quotes. Furthermore, the string produced by sqlite3_mprintf() is written into memory obtained from malloc() so that there is never a possibility of a buffer overflow.

You may find sqlite3_mprintf() so handy that you may want to create your own modified version of sqlite3_exec() that takes a variable number of arguments and substitutes them into the SQL string automatically. This is where sqlite3_vmprintf() comes in. It allows you to do that very thing. In fact, this is how the execute() function in the common library included with the examples is implemented (see Listing 6-3).

Listing 6-3. *Using sqlite3_vmprintf()*

```
int execute(sqlite3 *db, const char* sql, ...)
{
    char *err, *tmp;

    va_list ap;
    va_start(ap, sql);
    tmp = sqlite3_vmprintf(sql, ap);
    va_end(ap);

    int rc = sqlite3_exec(db, tmp, NULL, NULL, &err);

    if(rc != SQLITE_OK) {
        if (err != NULL) {
            fprintf(stdout, "execute() : Error %i : %s\n", rc, err);
            sqlite3_free(err);
        }
    }

    sqlite3_free(tmp);

    return rc;
}
```

The execute() function takes an arbitrary number of arguments and substitutes them into the sql argument according to its specification. Using this function, you can put together a properly formatted, parameterized SQL statement without having to do any string processing, as shown in the following example:

```
int cid = 1;
char* sql insert into episodes (cid, name) values (%i,'%q');
execute(db, sql, 1, "Here's trouble");
```

This simple wrapper function provides protection against buffer overflows, performs automatic character escaping, and handles memory management, in addition to all the other useful features of `sprintf()`—all credit due to `sqlite3_vmprintf()`.

The Get Table Query

The `sqlite3_get_table()` function returns an entire result set of a command in a single function call. Just as `sqlite3_exec()` wraps the prepared query API functions, allowing you to run them all at once, `sqlite3_get_table()` wraps `sqlite3_exec()` for commands that return data with just as much convenience. Using `sqlite3_get_table()`, you don't have to bother with the `sqlite3_exec()` callback function, thus making it easier to fetch records. `sqlite3_get_table()` has the following declaration:

```
int sqlite3_get_table(
  sqlite3*,              /* An open database */
  const char *sql,       /* SQL to be executed */
  char ***resultp,       /* Result written to a char *[] that this points to */
  int *nrow,             /* Number of result rows written here */
  int *ncolumn,          /* Number of result columns written here */
  char **errmsg          /* Error msg written here */
);
```

This function takes all of the records returned from the SQL statement in `sql` and stores them in the `resultp` argument using memory declared on the heap (using `malloc()`). The memory must be freed using `sqlite3_free_table()`, which takes the `resultp` pointer as its sole argument. The first record in `resultp` is actually not a record, but the names of the columns in the result set. Consider the code in Listing 6-4 (taken from `get_table.c`).

Listing 6-4. *Using sqlite3_get_table*

```
int main(int argc, char **argv)
{
    /* Connect to database, etc. */

    char *result[];
    sql = "select * from episodes;";
    rc = sqlite3_get_table(db, sql, &result, &nrows, &ncols, &zErr);

    /* Do something with data */

    /* Free memory */
    sqlite3_free_table(result)
}
```

If, for example, the result set returned is of the form

```
name               | id
--------------------
The Junior Mint    | 43
The Smelly Car     | 28
The Fusilli Jerry  | 21
```

then the format of the result array will be structured as follows:

```
result [0] = "name";
result [1] = "id";
result [2] = "The Junior Mint";
result [3] = "43";
result [4] = "The Smelly Car";
result [5] = "28";
result [6] = "The Fusilli Jerry";
result [7] = "21";
```

The first two elements contain the column headings of the result set. Therefore, you can think of the result set indexing as 1-based with respect to rows, but 0-based with respect to columns. An example may help clarify this. The code to print out each column of each row in the result set is shown in Listing 6-5.

Listing 6-5. *Iterating Through sqlite3_get_table() Results*

```
rc = sqlite3_get_table(db, sql, &result, &nrows, &ncols, &zErr);

for(i=0; i < nrows; i++) {
    for(j=0; j < ncols; j++) {
        /* the i+1 term skips over the first record,
        which is the column headers */
        fprintf(stdout, "%s", result[(i+1)*ncols + j]);
    }
}
```

Prepared Queries

As you saw in Chapter 5, SQLite's approach to executing SQL commands consists of the prepare, step, and finalize functions. This section covers all aspects of this process, including stepping through result sets, fetching records, and using parameterized queries.

The wrapper functions simply wrap all of these steps into a single function call, making it more convenient in some situations to run specific commands. Each query function provides its own way of getting at rows and columns. As a general rule, the more packaged the method is, the less control you have over execution and results. Therefore, prepared queries offer the most features, the most control, and the most information. Following that is `sqlite3_exec()`; following that is `sqlite3_get_table()`.

Prepared queries use a special group of functions to access field and column information from a row. You get column values using `sqlite3_column_xxx()`, where *xxx* represents the data type of the value to be returned (e.g., `int`, `double`, `blob`). You can retrieve data in whatever format

you like. You can also obtain the declared types of columns (as they are defined in the `CREATE TABLE` statement) and other miscellaneous metadata such as storage format and both associated table and database names. `sqlite3_exec()`, by comparison, provides only a fraction of this information through its callback function. The same is true with `sqlite3_get_table()`, which only includes the result set's column headers with the data.

In practice you will find that each query method has its uses. `sqlite3_exec()` is especially good for running commands that modify the database (`CREATE`, `DROP`, `INSERT`, `UPDATE`, and `DELETE`). One function call and it's done. Prepared queries are typically better for `SELECT` statements because they offer so much more information, more linear coding (no callback functions), and more control by using cursors to iterate over results.

As you'll recall from Chapter 5, prepared queries are performed in three basic steps: compilation, execution, and finalization. This process is illustrated in Figure 6-1.

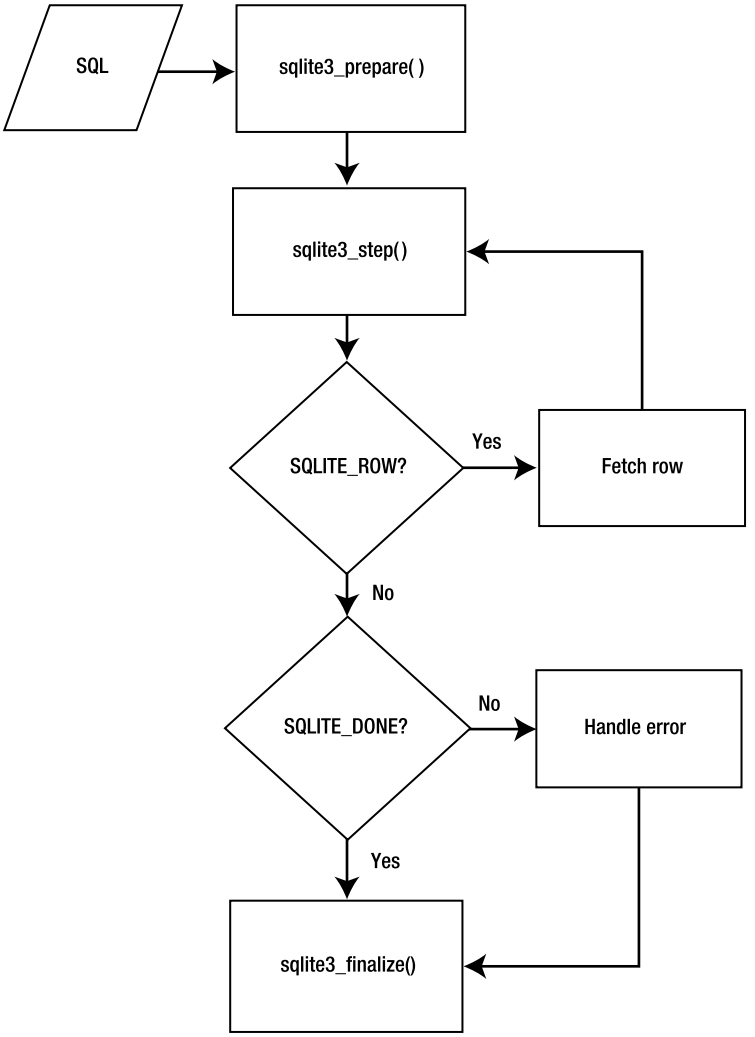

Figure 6-1. *Prepared query processing*

You compile the query with sqlite3_prepare(), execute it step by step using sqlite3_step(), and close it using sqlite3_finalize(), or you can reuse it using sqlite3_reset(). This process and the individual steps are all explained in detail in the following sections.

Compilation

Compilation, or preparation, takes a SQL statement and compiles it into byte code readable by the virtual database engine (VDBE). It is performed by sqlite3_prepare(), which is declared as follows:

```
int sqlite3_prepare(
  sqlite3 *db,            /* Database handle          */
  const char *zSql,       /* SQL text, UTF-8 encoded  */
  int nBytes,             /* Length of zSql in bytes. */
  sqlite3_stmt **ppStmt,  /* OUT: Statement handle    */
  const char **pzTail     /* OUT: Pointer to unused portion of zSql */
);
```

sqlite3_prepare() compiles the first SQL statement in the zSQL string (which can contain multiple SQL statements) into VDBE byte code. It allocates all the resources necessary to execute the statement and associates it along with the byte code into a single statement handle (also referred to as simply a statement), designated by the out parameter ppStmt, which is a sqlite3_stmt structure. From the programmer's perspective, this structure is little more than an opaque handle used to execute a SQL statement and obtain its associated records. However, this data structure contains the command's byte code, bound parameters, B-tree cursors, VDBE execution context, and any other data sqlite3_step() needs to manage the state of the query during execution.

sqlite3_prepare() does not affect the connection or database in any way. It does not start a transaction or get a lock. It works directly with the compiler, which simply prepares the query for execution. Statement handles are highly dependent on the database schema they were compiled with. If another connection alters the database schema, between the time you prepare a statement and the time you actually run it, your statement will expire. Your first call to sqlite3_step() with the statement will lead to a SQLITE_SCHEMA error, which is discussed later in the section "Errors and the Unexpected." Then you will have to finalize or reset the statement, recompile, and try again.

Execution

Once you prepare the query, the next step is to execute it using sqlite3_step(), declared as follows:

```
int sqlite3_step(sqlite3_stmt *pStmt);
```

sqlite3_step() takes the statement handle and talks directly to the VDBE. The VDBE reads the statement handle's byte code and steps through its instructions one by one to carry out the SQL statement. On the first call to sqlite3_step(), the VDBE obtains the requisite database lock needed to perform the command. If it can't get the lock, sqlite3_step() will return SQLITE_BUSY, if there is no busy handler installed. If one is installed, it will call that handler instead.

For SQL statements that don't return data, the first call to sqlite3_step() executes the command in its entirety, returning a result code indicating the outcome. For SQL statements

that do return data, such as SELECT, the first call to sqlite3_step() positions the statement's B-tree cursor on the first record. Subsequent calls to sqlite3_step() position the cursor on subsequent records in the result set. sqlite3_step() returns SQLITE_ROW for each record in the set until it reaches the end, whereupon it returns SQLITE_DONE, indicating that the cursor has reached the end of the set.

All other API functions related to data access use the statement's cursor to obtain information about the current record. For example, the sqlite3_column_*xxx*() functions all use the statement handle, specifically its cursor, to fetch the current record's fields.

Finalization and Reset

Once the statement has reached the end of execution, it must be finalized. You can either finalize or reset the statement using one of the following functions:

```
int sqlite3_finalize(sqlite3_stmt *pStmt);
int sqlite3_reset(sqlite3_stmt *pStmt);
```

sqlite3_finalize() will close out the statement. It frees resources and commits or rolls back any implicit transactions (if the connection is in autocommit mode), clearing the journal file and freeing the associated lock.

If you want to reuse the statement, you can do so using sqlite3_reset(). It will keep the compiled SQL statement (and any bound parameters), but commits any changes related to the current statement to the database. It also releases its lock and clears the journal file if autocommit is enabled. The primary difference between sqlite3_finalize() and sqlite3_reset() is that the latter preserves the resources associated with the statement so that it can be executed again, avoiding the need to call sqlite3_prepare() to compile the SQL command.

A Practical Example

Now that you've seen the whole process, let's go through an example. A simple, complete program using a prepared query is listed in Listing 6-6. It is taken from select.c in the examples.

Listing 6-6. *Using Prepared Queries*

```
int main(int argc, char **argv)
{
    int rc, i, ncols;
    sqlite3 *db;
    sqlite3_stmt *stmt;
    char *sql;
    const char *tail;

    rc = sqlite3_open("foods.db", &db);

    if(rc) {
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        exit(1);
    }
```

```
    sql = "select * from episodes;";

    rc = sqlite3_prepare(db, sql, strlen(sql), &stmt, &tail);

    if(rc != SQLITE_OK) {
        fprintf(stderr, "SQL error: %s\n", sqlite3_errmsg(db));
    }

    rc = sqlite3_step(stmt);
    ncols = sqlite3_column_count(stmt);

    while(rc == SQLITE_ROW) {

        for(i=0; i < ncols; i++) {
            fprintf(stderr, "'%s' ", sqlite3_column_text(stmt, i));
        }

        fprintf(stderr, "\n");

        rc = sqlite3_step(stmt);
    }

    sqlite3_finalize(stmt);
    sqlite3_close(db);

    return 0;
}
```

This example connects to the `foods.db` database, queries the `episodes` table, and prints out all columns of all records within it. Keep in mind this is a simplified example—there are a few other things we need to check for when calling `sqlite3_step()`, such as errors and busy conditions, but we will address them later.

Like `sqlite3_exec()`, `sqlite3_prepare()` can accept a string containing multiple SQL statements. However, unlike `sqlite3_exec()`, it will only process the first statement in the string. But it does make it easy for you to process subsequent SQL statements in the string by providing the `pzTail` out parameter. After you call `sqlite3_prepare()`, it will point this parameter (if provided) to the starting position of the next statement in the `zSQL` string. Using `pzTail`, processing a batch of SQL commands in a given string can be executed in a loop as follows:

```
while(sqlite3_complete(sql) {
    rc = sqlite3_prepare(db, sql, strlen(sql), &stmt, &tail);

    /* Process query results */

    /* Skip to next command in string. */
    sql = tail;
}
```

This example makes use of another API function not yet covered—sqlite3_complete(), which does as its name suggests: it takes a string and returns true (1) if there is at least one complete (but necessarily valid) SQL statement in it, and false otherwise.

Fetching Records

So far, you have seen how to obtain records and columns from sqlite3_exec() and sqlite3_get_table(). Prepared queries, by comparison, offer many more options when it comes to getting information from records in the database.

For a statement that returns records, the number of columns in the result set can be obtained using sqlite3_column_count() and sqlite3_data_count(), which are declared as follows:

```
int sqlite3_column_count(sqlite3_stmt *pStmt);
int sqlite3_data_count(sqlite3_stmt *pStmt);
```

sqlite3_column_count() returns the number of columns associated with a statement handle. You can call it on a statement handle before it is actually executed. If the query in question is not a SELECT statement, sqlite3_column_count() will return 0. Similarly, sqlite3_data_count() returns the number of columns for the current record, after sqlite3_step() returns SQLITE_ROW. This function will only work if the statement handle has an active cursor.

Getting Column Information

You can obtain the name of each column in the current record using sqlite3_column_name(), which is declared as follows:

```
const char *sqlite3_column_name( sqlite3_stmt*, /* statement handle */
                                 int iCol       /* column ordinal   */);
```

Similarly, you can get the associated storage class for each column using sqlite3_column_type(), which is declared as follows:

```
int sqlite3_column_type( sqlite3_stmt*, /* statement handle */
                         int iCol       /* column ordinal   */);
```

This function returns an integer value that corresponds to one of five storage class codes, defined as follows:

```
#define SQLITE_INTEGER  1
#define SQLITE_FLOAT    2
#define SQLITE_TEXT     3
#define SQLITE_BLOB     4
#define SQLITE_NULL     5
```

These are SQLite's native data types, or storage classes as described in Chapter 4. All data stored within a SQLite database is stored in one of these five forms, depending on its initial representation and the affinity of the column. For our purposes, the terms *storage class* and *data type* are synonymous. For more information on storage classes, see the sections "Storage Classes" and "Type Affinity" in Chapter 4.

You can obtain the declared data type of a column as it is defined in the table's schema using the sqlite3_column_decltype() function, which is declared as follows:

```
const char *sqlite3_column_decltype( sqlite3_stmt*, /* statement handle */
                                     int            /* column ordinal   */);
```

If a column in a result set does not correspond to an actual table column (say, for example, the column is the result of a literal value, expression, function, or aggregate), this function will return `NULL` as the declared type of that column. For example, suppose you have a table in your database defined as

```
CREATE TABLE t1(c1 INTEGER);
```

Then you execute the following query:

```
SELECT c1 + 1, 0 FROM t1;
```

In this case, `sqlite3_column_decltype()` will return `INTEGER` for the first column and `NULL` for the second.

In addition to the declared type, you can obtain other information on a column using the following functions:

```
const char *sqlite3_column_database_name(sqlite3_stmt *pStmt, int iCol);
const char *sqlite3_column_table_name(sqlite3_stmt *pStmt, int iCol);
const char *sqlite3_column_origin_name(sqlite3_stmt *pStmt, int iCol);
```

The first function will return the database associated with a column, the second its table, and the last function returns the column's actual name as defined in the schema. That is, if you assigned the column an alias in the SQL statement, `sqlite3_column_origin_name()` will return its actual name as defined in the schema. Note that these functions are only available if you compile SQLite with the `SQLITE_ENABLE_COLUMN_METADATA` preprocessor directive.

COLUMN METADATA

Detailed information about a column can be obtained independently from a query using the `sqlite3_table_column_metadata()` function, declared as follows:

```
int sqlite3_table_column_metadata(
    sqlite3 *db,                /* Connection handle                */
    const char *zDbName,        /* Database name or NULL            */
    const char *zTableName,     /* Table name                       */
    const char *zColumnName,    /* Column name                      */
    char const **pzDataType,    /* OUTPUT: Declared data type       */
    char const **pzCollSeq,     /* OUTPUT: Collation sequence name  */
    int *pNotNull,              /* OUTPUT: True if NOT NULL
                                           constraint exists        */
    int *pPrimaryKey,           /* OUTPUT: True if column part of PK */
    int *pAutoinc               /* OUTPUT: True if colums is
                                           auto-increment           */
);
```

This function is a combination of input and output parameters. It does not work from a statement handle, but rather from a combination of connection handle, database name, table name, and column name.

The optional database name refers to the logical name of an attached database (e.g., "main" or "temp"). If no database name is provided (the argument is `NULL`), then the function will search all attached databases for matching columns. Both table name and column name are required. The information for the matched column is provided to the memory locations of arguments 5 through 9. The memory for the `pzDataType` and `pzCollSeq` out parameters is valid only until the next API call. If no matching column can be found, then `sqlite3_table_column_metadata()` returns `SQLITE_ERROR`.

Getting Column Values

You can obtain the values for each column in the current record using the `sqlite3_column_xxx()` functions, which are of the general form

```
xxx sqlite3_column_xxx( sqlite3_stmt*, /* statement handle */
                        int iCol       /* column ordinal   */);
```

Here *xxx* is the data type you want the data represented in (e.g., `int`, `blob`, `double`, etc.). The complete list of the `sqlite3_column_xxx()` functions is as follows:

```
int sqlite3_column_int(sqlite3_stmt*, int iCol);
double sqlite3_column_double(sqlite3_stmt*, int iCol);
long long int sqlite3_column_int64(sqlite3_stmt*, int iCol);
const void *sqlite3_column_blob(sqlite3_stmt*, int iCol);
const unsigned char *sqlite3_column_text(sqlite3_stmt*, int iCol);
const void *sqlite3_column_text16(sqlite3_stmt*, int iCol);
```

For each function, SQLite converts the internal representation (storage class in the column) to the type specified in the function name. There are a number of rules SQLite uses to convert the internal data type representation to that of the requested type. These rules are listed in Table 6-1.

Table 6-1. *Column Type Conversion Rules*

Internal Type	Requested Type	Conversion
`NULL`	`INTEGER`	Result is 0.
`NULL`	`FLOAT`	Result is 0.0.
`NULL`	`TEXT`	Result is a `NULL` pointer.
`NULL`	`BLOB`	Result is a `NULL` pointer.
`INTEGER`	`FLOAT`	Convert from integer to float.
`INTEGER`	`TEXT`	Result is the ASCII rendering of the integer.
`INTEGER`	`BLOB`	Result is the ASCII rendering of the integer.
`FLOAT`	`INTEGER`	Convert from float to integer.

Table 6-1. *Column Type Conversion Rules (Continued)*

Internal Type	Requested Type	Conversion
FLOAT	TEXT	Result is the ASCII rendering of the float.
FLOAT	BLOB	Result is the ASCII rendering of the float.
TEXT	INTEGER	Use atoi().
TEXT	FLOAT	Use atof().
TEXT	BLOB	No change.
BLOB	INTEGER	Convert to TEXT and then use atoi().
BLOB	FLOAT	Convert to TEXT and then use atof().
BLOB	TEXT	Add a \000 terminator if needed.

Like the sqlite3_bind_*xxx*() functions described later, BLOBs require a little more work in that you must specify their length in order to copy them. For BLOB columns, you can get the actual length of the data using sqlite3_column_bytes(), which is declared as follows:

```
int sqlite3_column_bytes( sqlite3_stmt*, /* statement handle   */
                          int            /* column ordinal */);
```

Once you get the length, you can copy the binary data using sqlite3_column_blob(). For example, say the first column in the result set contains binary data. One way to get a copy of that data would be as follows:

```
int len = sqlite3_column_bytes(stmt,0);
void* data = malloc(len);
memcpy(data, len, sqlite3_column_blob(stmt,0));
```

A Practical Example

To help solidify all of these column functions, Listing 6-7 (taken from columns.c) illustrates using the functions we've described to retrieve column information and values for a simple SELECT statement.

Listing 6-7. *Obtaining Column Information*

```
int main(int argc, char **argv)
{
    int rc, i, ncols, id, cid;
    char *name, *sql;
    sqlite3 *db;
    sqlite3_stmt *stmt;

    sql = "select id, name from episodes";
    sqlite3_open("test.db", &db);
```

```
    setup(db);

    sqlite3_prepare(db, sql, strlen(sql), &stmt, NULL);

    ncols = sqlite3_column_count(stmt);
    rc = sqlite3_step(stmt);

    /* Print column information */
    for(i=0; i < ncols; i++) {
        fprintf(stdout, "Column: name=%s, storage class=%i, declared=%s\n",
                        sqlite3_column_name(stmt, i),
                        sqlite3_column_type(stmt, i),
                        sqlite3_column_decltype(stmt, i));
    }

    fprintf(stdout, "\n");

    while(rc == SQLITE_ROW) {
        id = sqlite3_column_int(stmt, 0);
        cid = sqlite3_column_int(stmt, 1);
        name = sqlite3_column_text(stmt, 2);
        if(name != NULL){
            fprintf(stderr, "Row:  id=%i, cid=%i, name='%s'\n", id,cid,name);
        } else {
            /* Field is NULL */
            fprintf(stderr, "Row:  id=%i, cid=%i, name=NULL\n", id,cid);
        }
        rc = sqlite3_step(stmt);
    }

    sqlite3_finalize(stmt);
    sqlite3_close(db);
    return 0;
}
```

This example connects to the database, selects records from the `episodes` table, and prints the column information and the fields for each row (using their internal storage class). Running the program produces the following output:

```
Column: name=id, storage class=1, declared=integer
Column: name=name, storage class=3, declared=text

Row:  id=1, name='The Dinner Party'
Row:  id=2, name='The Soup Nazi'
Row:  id=3, name='The Fusilli Jerry'
```

FINDING A STATEMENT'S CONNECTION

In practice you may find yourself writing code where some of your functions only have access to the statement handle, not the connection handle. If these functions encounter an error while working with the statement handle, they will not have a way to get error information from `sqlite3_errmsg()`, as it requires a connection handle to work. This is where `sqlite3_db_handle()` comes in handy. It is declared as follows:

```
int sqlite3_db_handle(sqlite3_stmt*);
```

Given a statement handle, `sqlite3_db_handle()` returns the associated connection handle. This way, you don't need to worry about having to pass the connection handle along with the statement handle everywhere you process query results.

Parameterized Queries

The API includes support for designating parameters in a SQL statement, allowing you to provide (or "bind") values for them at a later time. Bound parameters are used in conjunction with `sqlite3_prepare()`. For example, you could create a SQL statement like the following:

```
insert into foo values (?,?,?);
```

Then you can, for example, bind the integer value 2 to the first parameter (designated by the first ? character), the string value `'pi'` to the second parameter, and the double value 3.14 for the third parameter, as illustrated in the following code (taken from `parameters.c`):

```
const char* sql = "insert into foo values(?,?,?)";
sqlite3_prepare(db, sql, strlen(sql), &stmt, &tail);

sqlite3_bind_int(stmt,    1, 2);
sqlite3_bind_text(stmt,   2, "pi");
sqlite3_bind_double(stmt, 3, 3.14);

sqlite3_step(stmt);
sqlite3_finalize(stmt);
```

This generates and executes the statement

```
insert into foo values (2, 'pi', 3.14)
```

This process is illustrated in Figure 6-2. This particular method of binding uses *positional parameters* (as described in Chapter 5) where each parameter is designated by a question mark (`?`) character, and later identified by its index or relative position in the SQL statement.

Before delving into the other parameter methods, it is helpful to first understand the process by which parameters are defined, bound, and evaluated. When you write a parameterized statement such as the following, the parameters within it are identified when `sqlite3_prepare()` compiles the query:

```
insert into episodes (id,name) values (?,?)
```

sqlite3_prepare() recognizes that there are parameters in a SQL statement. Internally, it assigns each parameter a number to uniquely identify it. In the case of positional parameters, it starts with 1 for the first parameter found and uses sequential integer values for subsequent parameters. It stores this information in the resulting statement handle (sqlite3_stmt structure), which will then expect a specific number of values to be bound to the given parameters before execution. If you do notbind a value to a parameter, sqlite3_step() will use NULL for its value by default when the statement is executed.

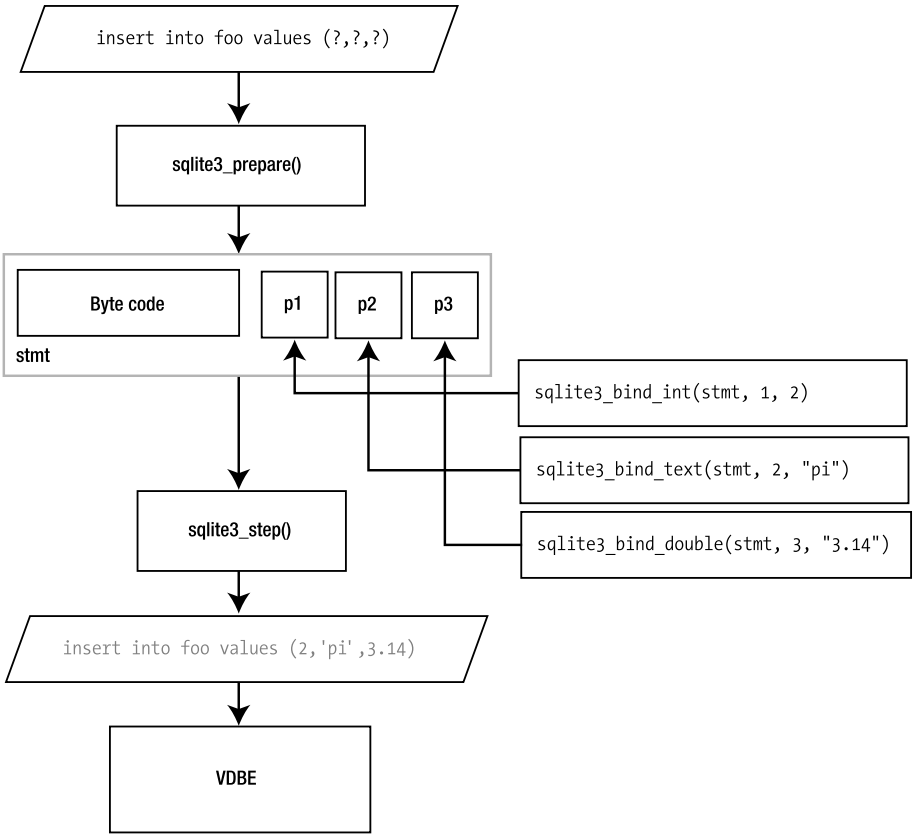

Figure 6-2. *SQL parameter binding*

After you prepare the statement, you then bind values to it. You do this using the sqlite3_bind_*xxx*() functions, which have the general form

```
sqlite3_bind_xxx( sqlite3_stmt*, /* statement handle  */
                  int i,         /* parameter number  */
                  xxx value      /* value to be bound */
                );
```

The *xxx* in the function name represents the data type of the value to bind. For example, to bind a double value to a parameter, you would use sqlite3_bind_double(), which is declared as follows:

```
sqlite3_bind_double(sqlite3_stmt* stmt, int i, double value);
```

The complete list of bind functions is as follows:

```
int sqlite3_bind_int(sqlite3_stmt*, int, int);
int sqlite3_bind_double(sqlite3_stmt*, int, double);
int sqlite3_bind_int64(sqlite3_stmt*, int, long long int);
int sqlite3_bind_null(sqlite3_stmt*, int);

int sqlite3_bind_blob( sqlite3_stmt*, int, const void*,
                       int n, void(*)(void*));
int sqlite3_bind_text( sqlite3_stmt*, int, const char*,
                       int n, void(*)(void*));
int sqlite3_bind_text16( sqlite3_stmt*, int, const void*,
                         int n, void(*)(void*));
```

In general, the bind functions can be divided into two classes, one for scalar values (`int`, `double`, `int64`, and `NULL`), and the other for arrays (`blob`, `text`, and `text16`). They differ only in that the array bind functions require a length argument and a pointer to a cleanup function. Also, `sqlite3_bind_text()` automatically escapes quote characters like `sqlite3_mprintf()`. Using the `BLOB` variant, the array bind function has the following declaration:

```
int sqlite3_bind_blob( sqlite3_stmt*,   /* statement handle       */
                       int,             /* ordinal               */
                       const void*,     /* pointer to blob data  */
                       int n,           /* length (bytes) of data */
                       void(*)(void*)); /* cleanup hander        */
```

There are two predefined values for the cleanup handler provided by the API that have special meanings, defined as follows:

```
#define SQLITE_STATIC      ((void(*)(void *))0)
#define SQLITE_TRANSIENT   ((void(*)(void *))-1)
```

Each value designates a specific cleanup action. `SQLITE_STATIC` tells the array bind function that the array memory resides in unmanaged space, so SQLite does not attempt to clean it up. `SQLITE_TRANSIENT` tells the bind function that the array memory is subject to change, so SQLite makes its own private copy of the data, which it automatically cleans up when the statement is finalized. The third option is to provide a pointer to your own cleanup function, which must be of the form

```
void cleanup_fn(void*)
```

If provided, SQLite will call your cleanup function, passing in the array memory when the statement is finalized.

■ **Note** Bound parameters remain bound throughout the lifetime of the statement handle. They remain bound even after a call to `sqlite3_reset()` and are only freed when the statement is finalized (by calling `sqlite3_finalize()`).

Once you have bound all of the parameters you want, you can then execute the statement. You do this using the next function in the sequence: `sqlite3_step()`. `sqlite3_step()` will take the bound values, substitute them for the parameters in the SQL statement, and then begin executing the command.

Also, as a matter of convenience, bindings can be transferred from one statement to another using `sqlite3_transfer_bindings()`, which is declared as follows:

```
int sqlite3_transfer_bindings( sqlite3_stmt*,  /* source statement */
                               sqlite3_stmt*); /* dest statement   */
```

This function is useful if you want to prepare the same query using another statement, without having to repeat all of the binding steps. This function is especially useful for dealing with `SQLITE_SCHEMA` errors, described later.

Now that you understand the binding process, the four parameter-binding methods differ only by

- The way in which parameters are represented in the SQL statement (using a positional parameter, explicitly defined parameter number, or alphanumeric parameter name)
- The way in which parameters are assigned numbers

For positional parameters, `sqlite3_prepare()` assigns numbers using sequential integer values starting with 1 for the first parameter. In the previous example, the first ? parameter is assigned 1, and the second ? parameter is assigned 2. With positional parameters, it is your job to keep track of which number corresponds to which parameter (or question mark) in the SQL statement and correctly specify that number in the bind functions.

Numbered Parameters

Numbered parameters, on the other hand, allow you to specify your own numbers for parameters, rather than use an internal sequence. The syntax for numbered parameters uses a question mark followed by the parameter number. Take, for example, the following piece of code (taken from `parameters.c`):

```
name = "Mackinaw Peaches";
sql  = "insert into episodes (id, cid, name) "
       "values (?100,?100,?101)";

rc = sqlite3_prepare(db, sql, strlen(sql), &stmt, &tail);

if(rc != SQLITE_OK) {
    fprintf(stderr, "sqlite3_prepare() : Error: %s\n", tail);
    return rc;
}
```

```
sqlite3_bind_int(stmt, 100, 10);
sqlite3_bind_text(stmt, 101, name, strlen(name), SQLITE_TRANSIENT);
sqlite3_step(stmt);
sqlite3_finalize(stmt);
```

This example uses 100 and 101 for its parameter numbers. Parameter number 100 has the integer value 10 bound to it. Parameter 101 has the string value `'Mackinaw Peaches'` bound to it. Note how numbered parameters come in handy when you need to bind a single value in more than one place in a SQL statement. Consider the values part of the previous SQL statement:

```
insert into episodes (id, cid, name) values (?100,?100,?101)";
```

Parameter 100 is being used twice—once for `id` and again for `cid`. Thus, numbered parameters save time when you need to use a bound value in more than one place.

■ **Note** When using numbered parameters in a SQL statement, keep in mind that the allowable range consists of the integer values 1–999.

Named Parameters

The third parameter binding method is using named parameters. Whereas you can assign your own numbers using numbered parameters, you can assign alphanumeric names with named parameters. Likewise, as numbered parameters are prefixed with a question mark (`?`), you identify named parameters by prefixing a colon (`:`) to the parameter name. Consider the following code snippet (taken from `parameters.c`):

```
name = "Mackinaw Peaches";
sql = "insert into episodes (id, cid, name) values (:cosmo,:cosmo,:newman)";

rc = sqlite3_prepare(db, sql, strlen(sql), &stmt, &tail);

sqlite3_bind_int( stmt,
                  sqlite3_bind_parameter_index(stmt, ":cosmo"), 10);

sqlite3_bind_text( stmt,
                   sqlite3_bind_parameter_index(stmt, ":newman"),
                   name,
                   strlen(name), SQLITE_TRANSIENT );

sqlite3_step(stmt);
sqlite3_finalize(stmt);
```

This example is identical to the previous example using numbered parameters, except it uses two named parameters called `cosmo` and `newman` instead. Like positional parameters, named parameters are automatically assigned numbers by `sqlite3_prepare()`. While the numbers assigned to each parameter are unknown, you can resolve them using `sqlite3_bind_parameter_index()`, which takes a parameter name and returns the corresponding

parameter number. This is the number you use to bind the value to its parameter. All in all, named parameters mainly help with legibility more than anything else.

PARAMETER INDEXES

While the function `sqlite3_bind_parameter_index()` seems to refer to a parameter number as an *index*, for all intents and purposes the two terms (number and index) are synonymous. I have settled on the term *number* rather than *index* for parameter positions as a result of some discussion on the SQLite mailing list, which has suggested that *number* is perhaps a better description. The term *index* was used when parameter binding was limited only to positional parameters. With the addition of numbered parameters, referring to parameter positions as (sequential) indexes can be somewhat misleading.

Tcl Parameters

The final parameter scheme is called Tcl parameters and is specific more to the Tcl extension than it is to the C API. Basically, it works identically to named parameters except that rather than using alphanumeric values for parameter names, it uses Tcl variable names. In the Tcl extension, when the Tcl equivalent of `sqlite3_prepare()` is called, the Tcl extension automatically searches for Tcl variables with the given parameter names in the active Tcl program environment and binds them to their respective parameters. Despite its current application in the Tcl interface, nothing prohibits this same mechanism from being applied to other language interfaces, which can in turn implement the same feature. In this respect, referring to this parameter method solely as Tcl parameters may be a bit of a misnomer. The Tcl extension just happened to be the first application that utilized this method. Basically, the Tcl parameter syntax does little more than provide an alternate syntax to named parameters—rather than prefixing the parameters with a colon (`:`), it uses a dollar sign (`$`).

Errors and the Unexpected

Up to now, we have looked at the API from a rather optimistic viewpoint, as if nothing could ever go wrong. But things do go wrong, and there is a part of the API devoted to that. The three things you always have to guard against in your code are errors, busy conditions, and my personal favorite: schema changes.

Handling Errors

Many of the API functions return integer result codes. That means they can potentially return error codes of some sort. The most common functions to watch are typically the most frequently used, such as `sqlite3_open()`, `sqlite3_prepare()` and friends, as well as `sqlite3_exec()`. You should always program defensively and review every API function (documented in Appendix B) before you use it to ensure that you deal with every possible error condition that can arise. There are about 23 different errors defined in the API. Only a fraction of them will really matter to your application in practice. All of the SQLite return codes are listed in Table 6-2. All of the API functions that can return them are listed as follows:

```
sqlite3_bind_xxx()
sqlite3_close()
sqlite3_create_collation()
sqlite3_collation_needed()
sqlite3_create_function()
sqlite3_prepare()
sqlite3_exec()
sqlite3_finalize()
sqlite3_get_table()
sqlite3_open()
sqlite3_reset()
sqlite3_step()
sqlite3_transfer_bindings()
```

You can get extended information on a given error using `sqlite3_errmsg()`, which is declared as follows:

```
const char *sqlite3_errmsg(sqlite3*);
```

It takes a connection handle as its only argument and returns the most recent error resulting from an API call on that connection. If no error has been encountered, it returns the string "not an error".

Table 6-2. *SQLite Return Codes*

Code	Description
`SQLITE_OK`	The operation was successful.
`SQLITE_ERROR`	General SQL error or missing database. It may be possible to obtain more error information depending on the error condition (`SQLITE_SCHEMA`, for example).
`SQLITE_PERM`	Access permission denied. Cannot read or write to the database file.
`SQLITE_ABORT`	A callback routine requested an abort.
`SQLITE_BUSY`	The database file is locked.
`SQLITE_LOCKED`	A table in the database is locked.
`SQLITE_NOMEM`	A call to `malloc()` has failed within a database operation.
`SQLITE_READONLY`	An attempt was made to write to a read-only database.
`SQLITE_INTERRUPT`	Operation was terminated by `sqlite3_interrupt()`.
`SQLITE_IOERR`	Some kind of disk I/O error occurred.
`SQLITE_CORRUPT`	The database disk image is malformed. This will also occur if an attempt is made to open a non-SQLite database file as a SQLite database.
`SQLITE_FULL`	Insertion failed because the database is full. There is no more space on the file system or the database file cannot be expanded.

Table 6-2. *SQLite Return Codes*

Code	Description
SQLITE_CANTOPEN	SQLite was unable to open the database file.
SQLITE_PROTOCOL	The database is locked or there has been a protocol error.
SQLITE_EMPTY	(Internal only) The database table is empty.
SQLITE_SCHEMA	The database schema has changed.
SQLITE_CONSTRAINT	Abort due to constraint violation. This constant is returned if the SQL statement would have violated a database constraint (such as attempting to insert a value into a unique index that already exists in the index).
SQLITE_MISMATCH	Data type mismatch. An example of this is an attempt to insert non-integer data into a column labeled INTEGER PRIMARY KEY. For most columns, SQLite ignores the data type and allows any kind of data to be stored. But an INTEGER PRIMARY KEY column is only allowed to store integer data.
SQLITE_MISUSE	Library was used incorrectly. This error might occur if one or more of the SQLite API routines is used incorrectly. Examples of incorrect usage include calling sqlite3_exec() after the database has been closed using sqlite3_close() or calling sqlite3_exec() with the same database pointer simultaneously from two separate threads.
SQLITE_NOLFS	Uses OS features not supported on host. This value is returned if the SQLite library was compiled with large file support (LFS) enabled but LFS isn't supported on the host operating system.
SQLITE_AUTH	Authorization denied. This occurs when a callback function installed using sqlite3_set_authorizer() returns SQLITE_DENY.
SQLITE_ROW	sqlite3_step() has another row ready.
SQLITE_DONE	sqlite3_step() has finished executing.

While it is very uncommon outside of embedded systems, one of the most critical errors you can encounter is SQLITE_NOMEM, which means that no memory can be allocated on the heap (e.g., malloc() failed). Normally, after a single malloc() call fails, the SQLite library refuses to function (all major calls can return SQLITE_NOMEM). If the application is able to recover from an out-of-memory condition, it may still be possible to restore the state of the SQLite library using sqlite3_global_recover(), which is declared as follows:

```
int sqlite3_global_recover();
```

This function restores the library state so that it can be used again. You must finalize or reset all active statements (sqlite3_stmt pointers) before calling this function. Otherwise it will return SQLITE_BUSY. This function will also return SQLITE_ERROR if you are using any in-memory databases, either as a main or TEMP. (The TEMP database is where temporary data is stored—see ATTACH DATABASE in Appendix A for details.) In either case, SQLite will not reset the library and it will remain unusable.

■ **Caution** `sqlite3_global_recover()` is not thread safe. Calling it from within a threaded application when threads other than the caller have used SQLite is dangerous and will almost certainly result in malfunctions. SQLite includes memory management functions specifically for threads. These are covered later in the section "Threads and Memory Management."

The `sqlite3_global_recover()` function is aimed at embedded applications where memory is scarcer than normal applications. It can be omitted from SQLite at compile time by defining the `SQLITE_OMIT_GLOBALRECOVER` preprocessor directive.

Handling Busy Conditions

Two important functions related to processing queries are `sqlite3_busy_handler()` and `sqlite3_busy_timeout()`. If your program uses a database on which there are other active connections, odds are it will eventually have to wait for a lock, and therefore will have to deal with `SQLITE_BUSY`. Whenever you call an API function that causes SQLite to seek a lock and SQLite is unable to get it, the function will return `SQLITE_BUSY`. There are three ways to deal with this:

- Handle `SQLITE_BUSY` yourself, either by rerunning the statement or taking some other action.
- Have SQLite call a busy handler.
- Ask SQLite to wait (block or sleep) for some period of time for the lock to clear.

The last option involves using `sqlite3_busy_timeout()`. This function tells SQLite how long to wait for a lock to clear before returning `SQLITE_BUSY`. While it can ultimately result in you still having to handle `SQLITE_BUSY`, in practice setting this value to a sufficient period of time (say 30 seconds) usually provides enough time for even the most intensive transaction to complete. Nevertheless, you should still have some contingency plan in place to handle `SQLITE_BUSY`.

User-Defined Busy Handlers

The second option entails using `sqlite3_busy_handler()`. This function provides a way to call a user-defined function rather than blocking or returning `SQLITE_BUSY` right away. It's declared as follows:

```
int sqlite3_busy_handler(sqlite3*, int(*)(void*,int), void*);
```

The second argument is a pointer to a function to be called as the busy handler, and the third argument is a pointer to application-specific data to be passed as the first argument to the handler. The second argument to the busy handler is the number of prior calls made to the handler for the same lock.

Such a handler might call `sleep()` for a period to wait out the lock, or it may send some kind of notification. It may do whatever you like, as it is yours to implement. Be warned, though, that registering a busy handler does not guarantee that it will always be called. As mentioned in Chapter 5, SQLite will forego calling a busy handler for a connection if it perceives a deadlock might result. Specifically, if your connection in `SHARED` is interfering with another connection in

RESERVED, SQLite will not invoke your busy handler, hoping you will take the hint. In this case, you are trying to write to the database from SHARED (starting the transaction with BEGIN) when you really should be starting from RESERVED (starting the transaction with BEGIN IMMEDIATE).

The only restriction on busy handlers is that they may not close the database. Closing the database from within a busy handler can delete critical data structures out from under the executing query and result in crashing your program.

Advice

All things considered, the best route may be to set the timeout to a reasonable value and then take some precaution if and when you receive a SQLITE_BUSY value. In general, if you are going to write to the database, start in RESERVED. If you don't do this, then the next best thing is to install a busy handler, set the timeout to a known value, and if SQLite returns SQLITE_BUSY, check the response time. If the time is less than the busy handler's delay, then SQLite is telling you that your query (and connection) is preventing a writer from proceeding. If you want to write to the database at this point, you should finalize or reset, then reexecute the statement, this time starting with BEGIN IMMEDIATE.

Handling Schema Changes

Whenever a connection changes the database schema, all other prepared statements that were compiled before the change are invalidated. The result is that the first call to sqlite3_step() for such statements returns SQLITE_ERROR. From a locking standpoint, the schema change occurs between the time a reader calls sqlite3_prepare() to compile a statement and calling sqlite3_step() to execute it.

When this happens, the only course of action for you is to finalize or reset the query and start over. However, you must confirm that the error is in fact due to a schema change. To do so, whenever sqlite3_step() returns SQLITE_ERROR, you should call sqlite3_reset() to see if it returns SQLITE_SCHEMA. Another way to check for this condition is to call sqlite3_expired(), which takes the statement handle as its only argument.

■ **Note** It's important that you understand that when a schema change occurs, sqlite3_step() *never* returns SQLITE_SCHEMA directly. It always returns SQLITE_ERROR. You then have to call sqlite3_finalize(), sqlite3_reset(), or sqlite3_expired() to determine if the error was due to SQLITE_SCHEMA. You should always check for this condition whenever you use sqlite3_step().

Several events can cause SQLITE_SCHEMA errors:

- Detaching databases
- Modifying or installing user-defined functions or aggregates
- Modifying or installing user-defined collations
- Modifying or installing authorization functions
- Vacuuming the database

The reason the SQLITE_SCHEMA condition exists ultimately relates to the VDBE. When a connection changes the schema, other compiled queries may have VDBE code that points to database objects that no longer exist, or are in a different location in the database. Rather than running the risk of a bizarre runtime error later, SQLite invalidates all statements that have been compiled but not executed. They must be recompiled.

SQLITE_SCHEMA can only occur on the first call to sqlite3_step(). This is because sqlite3_step(), when successful, always gets a lock on the database. Once you have a lock on the database, it is impossible for any other connection to write to or alter the database. So if the first call is successful, you are guaranteed that all subsequent calls to sqlite3_step() will *not* encounter SQLITE_SCHEMA. One possible way provided by the SQLite FAQ to handle SQLITE_SCHEMA is shown in Listing 6-8.

Listing 6-8. *Handling SQLITE_SCHEMA*

```
int rc;
sqlite3_stmt *pStmt;
char zSql[] = "SELECT .....";

do {
  /* Compile the statement from SQL. Assume success. */
  sqlite3_prepare(pDb, zSql, -1, &pStmt, 0);

  while( SQLITE_ROW==sqlite3_step(pStmt) ) {
      /* Do something with the row of available data */
  }

  /* Finalize the statement. If a SQLITE_SCHEMA error has
  ** occurred, then the above call to sqlite3_step() will have
  ** returned SQLITE_ERROR. sqlite3_finalize() will return
  ** SQLITE_SCHEMA. In this case the loop will execute again.
  */
  rc = sqlite3_finalize(pStmt);
} while( rc==SQLITE_SCHEMA );
```

A possible variation is to take into consideration bound parameters. These parameters can be automatically transferred to the new compiled query using sqlite3_transfer_bindings(), as shown in Listing 6-9.

Listing 6-9. *Handling SQLITE_SCHEMA with sqlite3_transfer_bindings()*

```
int rc, processed, skip;
sqlite3_stmt *pStmt = NULL;
sqlite3_stmt *plastStmt = NULL;
char zSql[] = "SELECT .....";

do {
    sqlite3_prepare(pDb, zSql, -1, &pStmt, 0);
```

```
    /* If there was a lastStmt, transfer bindings from it */
    if(plastStmt != NULL){
           sqlite3_transfer_bindings(plastStmt, pStmt);
    }

    /* Keep track of the current stmt */
    plastStmt = pStmt;

    while( SQLITE_ROW==sqlite3_step(pStmt) ) {
        /* Do something with the row of available data */
        processed++;
    }

    rc = sqlite3_finalize(pStmt);

} while( rc==SQLITE_SCHEMA );
```

Another option for checking for `SQLITE_SCHEMA` without finalizing (destroying) the query is to call `sqlite3_reset()` instead of `sqlite3_finalize()`. This will provide you with the `SQLITE_SCHEMA` error if it exists. However, if the error is not schema related, you avoid the cost of having to recompile the query since `sqlite3_reset()` leaves the query in a state where it can still be executed.

TRACING SQL

If you are having a hard time figuring out exactly what your program is doing with the database, you can track what SQL statements it has executed using `sqlite3_trace()`. Its declaration is as follows:

```
void *sqlite3_trace(sqlite3*, void(*xTrace)(void*,const char*), void*);
```

This function is analogous to putting a wiretap on a connection. You can use it to generate a log file of all SQL executed on a given connection as well provide helpful debugging information. Every SQL statement that is processed is passed to the callback function specified in the second argument. SQLite passes the data provided in the third argument of `sqlite3_trace()` to the first argument of the callback function.

Operational Control

The API provides several functions you can use to monitor and/or manage SQL commands at compile time and runtime. These functions allow you to install callback functions with which to monitor and control various database events as they happen.

Commit Hooks

The `sqlite3_commit_hook()` function allows you to monitor when transactions commit on a given connection. It is declared as follows:

```
void *sqlite3_commit_hook( sqlite3 *cnx,                /* database handle   */
                           int(*xCallback)(void *data), /* callback function */
                           void *data);                 /* application data  */
```

This function registers the callback function `xCallback`, which will be invoked whenever a transaction commits on the connection given by `cnx`. The third argument (`data`) is a pointer to application-specific data, which SQLite passes to the callback function. If the callback function returns a non-zero value, then the commit is converted into a rollback.

Passing a `NULL` value in for the callback function effectively disables the currently registered callback (if any). Also, only one callback can be registered at a time for a given connection. The return value for `sqlite3_commit_hook()` is `NULL` unless another callback function was previously registered, in which case the previous `data` value is returned.

■ **Note** The `sqlite3_commit_hook()` function is currently marked as experimental and is therefore subject to change. However, according SQLite's author, this is extremely unlikely as this function has been in the API for some time now.

Rollback Hooks

Rollback hooks are similar to commit hooks except that they watch for rollbacks for a given connection. Rollback hooks are registered with the following function:

```
void *sqlite3_rollback_hook(sqlite3 *cnx, void(*xCallback)(void *data), void *data);
```

This function registers the callback function `xCallback`, which will be invoked in the event of a rollback on `cnx`, whether by an explicit `ROLLBACK` command or an implicit error or constraint violation. The callback is *not* invoked if a transaction is automatically rolled back due to the database connection being closed. The third argument (`data`) is a pointer to application-specific data, which SQLite passes to the callback function.

As in `sqlite3_commit_hook()`, each time you call `sqlite3_rollback_hook()`, the new callback function you provide will replace any currently registered callback function. If a callback function was previously registered, `sqlite3_rollback_hook()` returns the previous `data` argument.

Update Hooks

The `sqlite3_update_hook()` is used to monitor all `UPDATE`, `INSERT`, and `DELETE` operations on rows for a given database connection. It has the following form:

```
void *sqlite3_update_hook(
  sqlite3 *cnx,
  void(*)(void *, int, char const*, char const*, sqlite_int64),
  void *data);
```

The first argument of the callback function is a pointer to application-specific data, which you provide in the third argument. The callback function has the following form:

```
void callback ( void * data,
                int operation_code,
                char const *db_name,
                char const *table_name,
                sqlite_int64 rowid),
```

The operation_code argument corresponds to SQLITE_INSERT, SQLITE_UPDATE, and SQLITE_DELETE for INSERT, UPDATE, and DELETE operations, respectively. The third and fourth arguments correspond to the database name and table name the operation took place on. The final argument is the ROWID of the affected row. The callback is not invoked for operations on system tables (e.g., sqlite_master and sqlite_sequence). The return value is a pointer to the previously registered callback function's data argument, if it exists.

Authorizer Functions

Perhaps the most powerful event filter is sqlite3_set_authorizer(). It allows you to monitor and control queries as they are compiled. This function is declared as follows:

```
int sqlite3_set_authorizer(
  sqlite3*,
  int (*xAuth)( void*,int,
                const char*, const char*,
                const char*,const char*),
  void *pUserData
);
```

This routine registers a callback function that serves as an authorization function. SQLite will invoke the callback function at statement compile time (not at execution time) for various database events. The intent of the function is to allow applications to safely execute user-supplied SQL. It provides a way to restrict such SQL from certain operations (e.g., anything that changes the database) or to deny access to specific tables or columns within the database.

For clarity, the form of the authorization callback function is as follows:

```
int auth( void*,         /* user data      */
          int,           /* event code     */
          const char*,   /* event specific */
          const char*,   /* event specific */
          const char*,   /* database name  */
          const char*    /* trigger or view name */ );
```

The first argument is a pointer to application-specific data, which is passed in on the fourth argument of sqlite3_set_authorizer(). The second argument to the authorization function will be one of the defined constants listed in Table 6-3. These values signify what kind of operation is to be authorized. The third and fourth arguments to the authorization function are specific to the event code. These arguments are listed with their respective event codes in Table 6-3.

The fifth argument is the name of the database (“main”, “temp”, etc.) if applicable. The sixth argument is the name of the innermost trigger or view that is responsible for the access attempt or NULL if this access attempt is directly from top-level SQL.

The return value of the authorization function should be one of the constants SQLITE_OK, SQLITE_DENY, or SQLITE_IGNORE. The meaning of the first two values is consistent for all events—permit or deny the SQL statement. SQLITE_DENY will abort the entire SQL statement and generate an error.

The meaning of SQLITE_IGNORE is specific to the event in question. Statements that read or modify records generate SQLITE_READ or SQLITE_UPDATE events for each column the statement attempts to operate on. In this case, if the callback returns SQLITE_IGNORE, the column in question will be excluded from the operation. Specifically, attempts to read data from this column yield only NULL values, and attempts to update it will silently fail.

Table 6-3. *SQLite Authorization Events*

Event Code	Argument 3	Argument 4
SQLITE_CREATE_INDEX	Index name	Table name
SQLITE_CREATE_TABLE	Table name	NULL
SQLITE_CREATE_TEMP_INDEX	Index name	Table name
SQLITE_CREATE_TEMP_TABLE	Table name	NULL
SQLITE_CREATE_TEMP_TRIGGER	Trigger name	Table name
SQLITE_CREATE_TEMP_VIEW	View name	NULL
SQLITE_CREATE_TRIGGER	Trigger name	Table name
SQLITE_CREATE_VIEW	View name	NULL
SQLITE_DELETE	Table name	NULL
SQLITE_DROP_INDEX	Index name	Table name
SQLITE_DROP_TABLE	Table name	NULL
SQLITE_DROP_TEMP_INDEX	Index name	Table name
SQLITE_DROP_TEMP_TABLE	Table name	NULL
SQLITE_DROP_TEMP_TRIGGER	Trigger name	Table name
SQLITE_DROP_TEMP_VIEW	View name	NULL
SQLITE_DROP_TRIGGER	Trigger name	Table name
SQLITE_DROP_VIEW	View name	NULL
SQLITE_INSERT	Table name	NULL
SQLITE_PRAGMA	Pragma name	First argument or NULL
SQLITE_READ	Table name	Column name
SQLITE_SELECT	NULL	NULL
SQLITE_TRANSACTION	NULL	NULL
SQLITE_UPDATE	Table name	Column name
SQLITE_ATTACH	Filename	NULL
SQLITE_DETACH	Database name	NULL

To illustrate this, the following example (the complete source of which is in `authorizer.c`) will create a table `foo`, defined as follows:

```
create table foo(x int, y int, z int)
```

It registers an authorizer function, which will

- Block reads of column `z`
- Block updates to column `x`
- Monitor `ATTACH` and `DETACH` database events
- Log database events as they happen

This is a rather long example, which uses the authorizer function to filter many different database events, so for clarity I am going to break the code into pieces. The authorizer function has the general form shown in Listing 6-10.

Listing 6-10. *Example Authorizer Function*

```
int auth( void* x, int type,
          const char* a, const char* b,
          const char* c, const char* d )
{
    const char* operation = a;

    printf( "    %s ", event_description(type));

    /* Filter for different database events
    ** from SQLITE_TRANSACTION to SQLITE_INSERT,
    ** UPDATE, DELETE, ATTACH, etc. and either allow or deny
    ** them.
    */

    return SQLITE_OK;
}
```

The first thing the authorizer looks for is a change in transaction state. If it finds a change, it prints a message:

```
    if((a != NULL) && (type == SQLITE_TRANSACTION)) {
        printf(": %s Transaction", operation);
    }
```

Next the authorizer filters events that result in a schema change:

```
    switch(type) {
        case SQLITE_CREATE_INDEX:
        case SQLITE_CREATE_TABLE:
        case SQLITE_CREATE_TRIGGER:
        case SQLITE_CREATE_VIEW:
```

```
        case SQLITE_DROP_INDEX:
        case SQLITE_DROP_TABLE:
        case SQLITE_DROP_TRIGGER:
        case SQLITE_DROP_VIEW:
        {
            // Schema has been modified somehow.
            printf(": Schema modified");
        }
```

The next filter looks for read attempts (which are fired on a column-by-column basis). Here, all read attempts are allowed unless the column name is z, in which case the function returns SQLITE_IGNORE. This will cause SQLite to return NULL for any field in column z, effectively blocking access to its data.

```
if(type == SQLITE_READ) {
        printf(": Read of %s.%s ", a, b);

        /* Block attempts to read column z */
        if(strcmp(b,"z")==0) {
            printf("-> DENIED\n");
            return SQLITE_IGNORE;
        }
    }
```

Next come INSERT and UPDATE filters. All INSERT statements are allowed. However, UPDATE statements that attempt to modify column x are denied. This will *not* block the UPDATE statement from executing; rather, it will simply filter out any attempt to update column x.

```
    if(type == SQLITE_INSERT) {
        printf(": Insert records into %s ", a);
    }

    if(type == SQLITE_UPDATE) {
        printf(": Update of %s.%s ", a, b);

        /* Block updates of column x */
        if(strcmp(b,"x")==0) {
            printf("-> DENIED\n");
            return SQLITE_IGNORE;
        }
    }
```

Finally, the authorizer filters DELETE, ATTACH, and DETACH statements and simply issues notifications when it encounters them:

```
    if(type == SQLITE_DELETE) {
        printf(": Delete from %s ", a);
    }
```

```
    if(type == SQLITE_ATTACH) {
        printf(": %s", a);
    }

    if(type == SQLITE_DETACH) {
        printf("-> %s", a);
    }

    printf("\n");
    return SQLITE_OK;
}
```

The (abbreviated) program is implemented as follows. As with the authorizer function, I will break it into pieces. The first part of the program connects to the database and registers the authorization function:

```
int main(int argc, char **argv)
{
    sqlite3 *db, *db2;
    char *zErr, *sql;
    int rc;

    /**  Setup */

    /* Connect to test.db */
    rc = sqlite3_open("test.db", &db);

    /**  Authorize and test

    /* 1. Register the authorizer function */
    sqlite3_set_authorizer(db, auth, NULL);
```

Step 2 illustrates the transaction filter:

```
    /* 2. Test transactions events */

    printf("program : Starting transaction\n");
    sqlite3_exec(db, "BEGIN", NULL, NULL, &zErr);

    printf("program : Committing transaction\n");
    sqlite3_exec(db, "COMMIT", NULL, NULL, &zErr);
```

Step 3 tests schema modifications by creating the test table `foo`:

```
    /* 3. Test table events */

    printf("program : Creating table\n");
    sqlite3_exec(db, "create table foo(x int, y int, z int)", NULL, NULL, &zErr);
```

Step 4 tests read (`SELECT`) and write (`INSERT` / `UPDATE`) control. It inserts a test record, selects it, updates it, and selects it again to observe the results of the `UPDATE`.

```
/* 4. Test read/write access */
printf("program : Inserting record\n");
sqlite3_exec(db, "insert into foo values (1,2,3)", NULL, NULL, &zErr);

printf("program : Selecting record (value for z should be NULL)\n");
print_sql_result(db, "select * from foo");

printf("program : Updating record (update of x should be denied)\n");
sqlite3_exec(db, "update foo set x=4, y=5, z=6", NULL, NULL, &zErr);

printf("program : Selecting record (notice x was not updated)\n");
print_sql_result (db, "select * from foo");

printf("program : Deleting record\n");
sqlite3_exec(db, "delete from foo", NULL, NULL, &zErr);

printf("program : Dropping table\n");
sqlite3_exec(db, "drop table foo", NULL, NULL, &zErr);
```

Several things are going on here. The program selects all records in the table, one of which is column z. We should see in the output that column z's value is NULL. All other fields should contain data from the table. Next, the program attempts to update all fields, the most important of which is column x. The update should succeed, but the value in column x should be unchanged, as the authorizer denies it. This is confirmed on the following SELECT statement, which shows that all columns were updated except for column x, which is unchanged. The program then drops the foo table, which should issue a schema change notification from the previous filter.

Step 5 tests the ATTACH and DETACH database commands. The thing to notice here is how the authorizer function is provided with the name of the database and that you can distinguish the operations being performed under the attached database as opposed to the main database.

```
/* 5. Test ATTACH/DETACH */

/* Connect to test2.db */
rc = sqlite3_open("test2.db", &db2);

if(rc) {
    fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db2));
    sqlite3_close(db2);
    exit(1);
 }

sqlite3_exec(db2, "drop table foo2", NULL, NULL, &zErr);
sqlite3_exec(db2, "create table foo2(x int, y int, z int)",
                   NULL, NULL, &zErr);

printf("program : Attaching database test2.db\n");
sqlite3_exec(db, "attach 'test2.db' as test2", NULL, NULL, &zErr);
```

```
    printf("program : Selecting record from attached database test2.db\n");
    sqlite3_exec(db, "select * from foo2", NULL, NULL, &zErr);

    printf("program : Detaching table\n");
    sqlite3_exec(db, "detach test2", NULL, NULL, &zErr);
```

Again for clarity, I will break up the program output into pieces. Upon executing it, the first thing we see is the transaction filter catching changes in transaction state:

```
program : Starting transaction
    SQLITE_TRANSACTION : BEGIN Transaction
program : Committing transaction
    SQLITE_TRANSACTION : COMMIT Transaction
program : Starting transaction
    SQLITE_TRANSACTION : BEGIN Transaction
program : Aborting transaction
    SQLITE_TRANSACTION : ROLLBACK Transaction
```

Next we see a schema change notification as the result of creating the test table `foo`. The interesting thing to notice here is all the other events that transpire in the `sqlite_master` table as a result of creating the table:

```
program : Creating table
    SQLITE_INSERT : Insert records into sqlite_master
    SQLITE_CREATE_TABLE : Schema modified
    SQLITE_READ : Read of sqlite_master.name
    SQLITE_READ : Read of sqlite_master.rootpage
    SQLITE_READ : Read of sqlite_master.sql
    SQLITE_UPDATE : Update of sqlite_master.type
    SQLITE_UPDATE : Update of sqlite_master.name
    SQLITE_UPDATE : Update of sqlite_master.tbl_name
    SQLITE_UPDATE : Update of sqlite_master.rootpage
    SQLITE_UPDATE : Update of sqlite_master.sql
    SQLITE_READ : Read of sqlite_master.ROWID
    SQLITE_READ : Read of sqlite_master.name
    SQLITE_READ : Read of sqlite_master.rootpage
    SQLITE_READ : Read of sqlite_master.sql
    SQLITE_READ : Read of sqlite_master.tbl_name
```

Next the program inserts a record, which the authorizer detects:

```
program : Inserting record
    SQLITE_INSERT : Insert records into foo
```

Now here is where things get more interesting. We are going to be able to see the authorizer block access to individual columns. The program selects all records from the foo table. We see the SQLITE_SELECT event take place, followed by the subsequent SQLITE_READ events generated for each attempted access of each column in the SELECT statement. When it comes to column z, the authorizer denies access. Immediately following that, SQLite executes the statement and print_sql_result() prints the column information and rows for the result set:

```
program : Selecting record (value for z should be NULL)
    SQLITE_SELECT
    SQLITE_READ : Read of foo.x
    SQLITE_READ : Read of foo.y
    SQLITE_READ : Read of foo.z -> DENIED
  Column: x (1/int)
  Column: y (1/int)
  Column: z (5/(null))
  Record: '1' '2' '(null)'
```

Look at what goes on with column z. Its value is NULL, which confirms that the authorizer blocked access. But also look at the column information. While SQLite revealed the storage class of column z, it denied access to its declared type in the schema.

Next comes the update. Here we are interested in column x. The UPDATE statement will attempt to change every value in the record. But update of column x will be denied:

```
program : Updating record (update of x should be denied)
    SQLITE_UPDATE : Update of foo.x -> DENIED
    SQLITE_UPDATE : Update of foo.y
    SQLITE_UPDATE : Update of foo.z
```

To confirm this, the program then selects the record to show what happened. The UPDATE did execute, but column x was not changed. Even more interestingly, while z was updated (trust me, it was), the authorizer will not let us see its value:

```
program : Selecting record (notice x was not updated)
    SQLITE_SELECT
    SQLITE_READ : Read of foo.x
    SQLITE_READ : Read of foo.y
    SQLITE_READ : Read of foo.z -> DENIED
  Column: x (1/int)
  Column: y (1/int)
  Column: z (5/(null))
  Record: '1' '5' '(null)'
```

Next the program deletes the record and drops the table. The latter operation generates all sorts of events on the sqlite_master table, just as when the table was created:

```
program : Deleting record
    SQLITE_DELETE : Delete from foo
program : Dropping table
    SQLITE_DELETE : Delete from sqlite_master
    SQLITE_DROP_TABLE : Schema modified
    SQLITE_DELETE : Delete from foo
    SQLITE_DELETE : Delete from sqlite_master
    SQLITE_READ : Read of sqlite_master.tbl_name
    SQLITE_READ : Read of sqlite_master.type
    SQLITE_UPDATE : Update of sqlite_master.rootpage
    SQLITE_READ : Read of sqlite_master.rootpage
```

Finally, the program creates another database on a separate connection and then attaches it on the main connection. The main connection then selects records from a table in the attached database, and we can see how the authorizer reports these operations as happening in the attached database:

```
program : Attaching database test2.db
    SQLITE_ATTACH : test2.db
    SQLITE_READ : Read of sqlite_master.name
    SQLITE_READ : Read of sqlite_master.rootpage
    SQLITE_READ : Read of sqlite_master.sql
program : Selecting record from attached database test2.db
    SQLITE_SELECT
    SQLITE_READ : Read of foo2.x
    SQLITE_READ : Read of foo2.y
    SQLITE_READ : Read of foo2.z -> DENIED
program : Detaching table
    SQLITE_DETACH -> test2
```

As you can see, `sqlite3_set_authorizer()` and its event-filtering capabilities are quite powerful. It gives you a great deal of control over what the user can and cannot do on a given database. You can monitor events and, if you wish, stop them before they happen. Or, you can allow them to proceed, but impose specific restrictions on them. `sqlite3_set_authorizer()` can be a helpful tool for dealing with `SQLITE_SCHEMA` conditions. If you know you don't need any changes to your database schema in your application, you can simply install an authorizer function to deny any operations that attempt to modify the schema in certain ways.

■ **Caution** Keep in mind that denying schema changes in an authorizer is by no means a cure-all for `SQLITE_SCHEMA` events. You need to be careful about all of the implications of denying various changes to the schema. Just blindly blocking all events resulting in a schema change can restrict other seemingly legitimate operations such as `VACUUM`.

HELP FOR INTERACTIVE PROGRAMS

SQLite provides two functions that make it easier to work with interactive programs. The first is `sqlite3_interrupt()`, which is declared as follows:

```
void sqlite3_interrupt(sqlite3* /* connection handle */);
```

`sqlite3_interrupt()` causes any pending database operation on the given connection to abort and return at its earliest opportunity. This routine is design to be called in response to a user interrupt action such as clicking a Cancel button in a graphical application or pressing Ctrl+C in a command-line program. It is intended to make it easier to accommodate cases where the user wants a long query operation to halt immediately. For a command-line program, you might put this function in a signal handler, or an event handler in a graphical application.

Another function that can be useful for interactive programs is `sqlite3_progress_handler()`. It is somewhat related in functionality to the `sqlite3_interrupt()` function, but with a few twists. It is declared as follows:

```
void sqlite3_progress_handler( sqlite3*,      /* connection handle     */
                               int frq,       /* frequency of callback */
                               int(*)(void*), /* callback function     */
                               void*);        /* application data      */
```

This function installs a callback function that will be invoked during calls to `sqlite3_exec()`, `sqlite3_step()`, and `sqlite3_get_table()`. The intent of this function is to give applications a way to provide feedback to the user during long-running queries.

The second argument (`frq`) specifies the frequency of the callback in terms of VDBE instructions. That is, the progress callback (provided in the third argument) will be called for every `frq` VDBE instruction performed in query execution. If a call to `sqlite3_exec()`, `sqlite3_step()`, or `sqlite3_get_table()` requires less than `frq` instructions to be executed, then the progress callback will not be invoked. The fourth argument is a pointer to application-specific data, which is passed back to the progress handler (as its one and only argument). If `NULL` is provided for the callback function argument, then the currently installed progress handler (if any) is disabled.

If the progress callback returns a non-zero value, the current query will be immediately terminated and any database changes rolled back. If the query was part of a larger transaction, then the transaction is *not* rolled back and remains active. The `sqlite3_exec()` call in that case will return `SQLITE_ABORT`.

Threads

There are a few basic rules to follow when using SQLite in a multithreaded environment. The first thing to be aware of, as mentioned in Chapter 2, is that you have compiled SQLite with thread support.

Before SQLite version 3.3.1, the general rule of thumb for using SQLite with threads was that a connection handle (`sqlite3` structure) can only be created, used, and destroyed within a given thread. It could not be safely passed around to other threads. This seems to be due mainly to limitations in specific thread implementations of some operating systems. Starting with 3.3.1, you can pass connections around between threads as long as they are in the `UNLOCKED` state at the time of the transfer (the connection has no transaction open).

One other limitation that is somewhat thread-related pertains to the `fork()` system call on Unix. You should never try to pass a connection across a `fork()` call to a child process. It will not work correctly.

Shared Cache Mode

Starting with SQLite version 3.3.0, SQLite supports something called *shared cache mode*, which allows multiple connections in a single thread to use a common page cache, as shown in Figure 6-3. This feature is designed for embedded servers where a single thread can efficiently manage multiple database connections on behalf of other threads. The connections in this case share a single page cache as well as a different concurrency model. Because of the shared cache, the server thread can operate with significantly lower memory usage and better concurrency than if each thread were to manage its own connection. Normally, each connection allocates enough memory to hold 2,000 pages in the page cache. Rather than each thread consuming that much memory with its own connection, it shares that memory with other threads using a single page cache.

In this model, threads rely on a server thread to manage their database connections for them. A thread sends SQL statements to the server through some communication mechanism; the server executes them using the thread's assigned connection, and then sends the results back. The thread can still issue commands and control its own transactions; only its actual connection exists in, and is managed by, another thread.

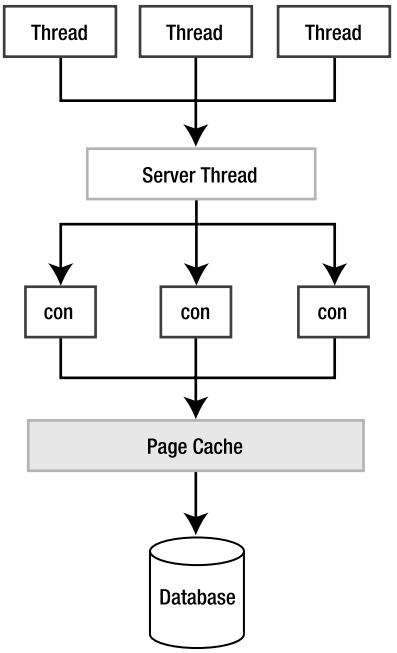

Figure 6-3. *The shared cache model*

Connections in shared cache mode use a different concurrency model and isolation level. Connections in a shared cache are strictly limited to the thread that created them. They cannot be passed around to other threads like normal connections can. Furthermore, connections in a shared cache can see what their kindred connections have changed. That is, there can be both multiple readers and an active writer in the database at the same time. That writer can write to the database—performing complete write transactions—without having to wait for other readers to clear. Therefore, data can change in the database within a reader's transaction. While SQLite normally runs in a serialized isolation level, meaning that readers and writers always see a consistent view of the database, in shared cache mode, readers are subject to the database changing underneath them.

There are some control measures in place to keep connections out of one another's way. By default, shared cache mode uses *table locks* to keep reader connections and writer connections separated. Table locks are not the same as database locks, and exist only within connections of the shared cache. Whenever a connection reads a table, SQLite first tries to get a read lock on it. The same is true for writing to a table. A connection may not write to a table that another connection has a read lock on, and vice versa. Furthermore, table locks are tied to their respective connection and last for the duration of its transaction.

A Practical Example

To make all this clear, let's go through an example. Say we have two threads, Thread A and Thread B. These threads communicate with a server thread, which we will represent as the program thread. Consider the first part of program in Listing 6-11, which represents the server thread starting up.

Listing 6-11. *Shared Cache Mode Example*

```
int main(int argc, char **argv)
{
    int rc, i, ncols;
    sqlite3 *db1;
    sqlite3 *db2;
    sqlite3_stmt *stmt;
    sqlite3_stmt *stmt2;
    char *sql;
    char *zErr;
    const char *tail;

    /* Enable shared cache mode */
    sqlite3_enable_shared_cache(1);
```

It's that simple. This program starts up and goes into shared cache mode by calling `sqlite3_enable_shared_cache()`. This function takes a single integer argument where 0 turns shared cache mode off for the thread, and any non-zero argument turns it on.

Next it simulates receiving a request from Thread A, which wants to connect to the database, query the `episodes` table, and put a cursor on the first row. This will get both a `SHARED` lock on the database and a table lock on `episodes`. The code for this step is shown in Listing 6-12.

Listing 6-12. *Thread A Initial Request*

```
sqlite3_open("foods.db", &db1);
sqlite3_exec(db1, "BEGIN", NULL, NULL, &zErr);
fprintf(stderr, "Thread A: Connected\n");
fprintf(stderr, "Thread A: BEGIN TRANSACTION\n");

sql = "select * from episodes;";
rc = sqlite3_prepare(db1, sql, strlen(sql), &stmt, &tail);
rc = sqlite3_step(stmt);

/* Send row back to A and wait for further instructions */
fprintf(stderr, "Thread A: SHARED lock on database, table lock on episodes\n");
```

The server thread connects to the database, manually starts a transaction, runs a `SELECT` on `episodes`, sits on the first record, and then pretends to send the first row back to Thread A.

Next, Thread B's request comes in. It tries to delete any rows in `foods` by the name of `'Whataburger'`. Next, it tries to insert a row with the name of `'Whataburger'`. After that, in order to illustrate table locks, it tries to insert a `'Whataburger'` row into `episodes`, which shouldn't work. It then closes its connection and exits. The code for this step is listed in Listing 6-13.

Listing 6-13. *Thread B Modifying the Database*

```
/* Request comes in from Thread B. Modify the foods table in auto-commit. */
sqlite3_open("foods.db", &db2);
fprintf(stderr, "Thread B: Connected\n");

sql = "delete from foods where name='Whataburger'";
rc = sqlite3_exec(db2, sql, NULL, NULL, &zErr);

if(rc != SQLITE_OK) {
    if (zErr != NULL) {
        fprintf(stderr, "Thread B: SQL error: %s\n", zErr);
        sqlite3_free(zErr);
    }
}
else {
    fprintf(stderr, "Thread B: TRANSACTION: ");
    fprintf(stderr, "Deleted any Whataburger rows from foods\n");
}
```

```
sql = "insert into foods values (NULL, 12, 'Whataburger')";
rc = sqlite3_exec(db2, sql, NULL, NULL, &zErr);

if(rc != SQLITE_OK) {
    if (zErr != NULL) {
        fprintf(stderr, "Thread B: SQL error: %s\n", zErr);
        sqlite3_free(zErr);
    }
}
else {
    fprintf(stderr, "Thread B: TRANSACTION: ");
    fprintf(stderr, "Inserted Whataburger row into foods\n");
}

/* Try to insert a record into episodes.
** This should not work b/c of Thread A's table lock. */
fprintf(stderr, "Thread B: Trying to insert Whataburger row into episodes\n");
sql = "insert into episodes values (NULL, 12, 'Whataburger')";
rc = sqlite3_exec(db2, sql, NULL, NULL, &zErr);

if(rc != SQLITE_OK) {
    if (zErr != NULL) {
        fprintf(stderr, "Thread B: SQL error: %s\n", zErr);
        sqlite3_free(zErr);
    }
}
else {
    fprintf(stderr, "Thread B: Inserted Whataburger row into episodes\n");
}

sqlite3_close(db2);
fprintf(stderr, "Thread B: Disconnected\n");
```

Finally, Thread A finalizes its initial statement and searches for the row that Thread B inserted into `foods`. The code for this step is shown in Listing 6-14.

Listing 6-14. *Thread A Reading Thread B's Modifications*

```
/* Close the last statement and look for Thread B's row. */
sqlite3_finalize(stmt);

fprintf(stderr, "Thread A: Querying foods for Whataburger record\n");
sql = "select id as foo, type_id, name from foods where name='Whataburger'";
rc = sqlite3_prepare(db1, sql, strlen(sql), &stmt, &tail);
rc = sqlite3_step(stmt);
ncols = sqlite3_column_count(stmt);
```

```
    fprintf(stderr, "Thread A: Results: ");
    for(i=0; i < ncols; i++) {
        fprintf(stderr, "'%s' ", sqlite3_column_origin_name(stmt, i));
        fprintf(stderr, "'%s' ", sqlite3_column_text(stmt, i));
    }
    fprintf(stderr, "\n", ncols);

    sqlite3_finalize(stmt);

    rc = sqlite3_exec(db1, "COMMIT", NULL, NULL, &zErr);
    fprintf(stderr, "Thread A: COMMIT\n");
    sqlite3_close(db1);
    fprintf(stderr, "Thread A: Exiting\n", ncols);
```

Running the program produces the following results:

```
Thread A: Connected
Thread A: BEGIN TRANSACTION
Thread A: SHARED lock on database, table lock on episodes
Thread B: Connected
Thread B: TRANSACTION: Deleted any Whataburger rows from foods
Thread B: TRANSACTION: Inserted Whataburger row into foods
Thread B: Trying to insert Whataburger row into episodes
Thread B: SQL error: database table is locked: episodes
Thread B: Disconnected
Thread A: Querying foods for Whataburger record
Thread A: Results: 'id' '413' 'type_id' '12' 'name' 'Whataburger'
Thread A: COMMIT
Thread A: Exiting
```

So what happened? In shared cache mode, both threads worked concurrently in the same database. Thread A got a table lock on `episodes`, and a `SHARED` lock on the database. Thread B came along and modified `foods`. It did two full write transactions—`DELETE` and `INSERT`—on `foods` in autocommit mode, each of which went all the way from `UNLOCKED` to `EXCLUSIVE` and back to `UNLOCKED`. When each transaction completed, its associated table lock on `foods` was released. Thread B couldn't modify `episodes`, however, because of Thread A's table lock. After Thread B exited, Thread A could read from `foods` and see Thread B's changes from within the same transaction.

Read Uncommitted Isolation Level

While table locks help keep connections in shared cache mode out of one another's way, it is possible for connections to elect not to be in the way at all. That is, a connection can choose to have a *read-uncommitted* isolation level by using the `read_uncommitted` pragma. If it is set to `true`, then the connection will not put read locks on the tables it reads. Therefore, another writer can actually change a table as the connection reads it. While this can lead to inconsistent

query results, it also means that a connection in read-uncommitted mode can neither block nor be blocked by any other connections.

It would be easy to modify the previous example to illustrate this kind of behavior. I will simply paraphrase it; you can code it if you want to. In this case, Thread A would go into read-committed isolation level, do a `SELECT..ORDER BY id`, and hold a cursor on the first row of the result set. Then Thread B would delete the second row of `episodes (id=2)`. Then you have Thread A position its cursor to the next row and see that it is in fact is the third row in `episodes` and not the second.

SCHEMA CHANGES IN SHARED CACHE MODE

There are additional things to consider in shared cache mode when a thread wants to modify the database schema. As it turns out, before SQLite issues a table lock, it first gets a read table lock on the `sqlite_master` table. In order for a thread to alter the schema, it must first get a write table lock on `sqlite_master`. It can only do this if there are no read locks on it from other tables. When a thread does get a write lock on `sqlite_master`, then no other threads are allowed to compile SQL statements during that time. They will get `SQLITE_SCHEMA` errors.

Threads and Memory Management

Since shared cache mode is about conserving memory, SQLite has several functions associated with threads and memory management. They allow you to specify an advisory heap limit—a soft heap limit—as well as manually initiate memory cleanups. These functions are as follows:

```
void sqlite3_soft_heap_limit(int N);
int sqlite3_release_memory(int N);
void sqlite3_thread_cleanup(void);
```

The `sqlite3_soft_heap_limit()` function sets the current soft heap limit of the calling thread to *N* bytes. If the thread's heap usage exceeds *N*, then SQLite automatically calls `sqlite3_release_memory()`, which attempts to free at least *N* bytes of memory from the caches of all database connections associated with the calling thread. The return value of `sqlite3_release_memory()` is the number of bytes actually freed.

If you use it, you must also set `sqlite3_soft_heap_limit()` back to zero (the default) prior to shutting down a thread or else it will leak memory. Alternatively, you can use `sqlite3_thread_cleanup()`, which ensures that a thread has released any thread-local storage it may have allocated. When the API is used properly, thread-local storage should be managed automatically, and you shouldn't need to call `sqlite3_thread_cleanup()`. However, it is provided as a precaution and a potential work-around for future thread-related memory leaks.

These routines are only available if you have enabled memory management by compiling SQLite with the `SQLITE_ENABLE_MEMORY_MANAGEMENT` preprocessor directive.

Summary

The core C API contains everything you need to process SQL commands and then some. It contains a variety of convenient query methods that are useful in different ways. These include `sqlite3_exec()` and `sqlite3_get_table()`, which allow you to execute commands in a single function call. The API includes many utility functions as well, allowing you to determine the number of affected records in a query, get the last inserted `ROWID` of an `INSERT` statement, trace SQL commands run on a connection, and conveniently format strings for SQL statements.

The `sqlite3_prepare()`, `sqlite3_step()`, and `sqlite3_finalize()` methods provide you with a lot of flexibility and control over statement execution through the use of statement handles. Statement handles provide more detailed row and column information, the convenience of bound parameters, as well as the ability to reuse prepared statements, avoiding the overhead of query compilation.

SQLite provides a variety of ways to manage runtime events through the use of event filters. Its commit, rollback, and update hooks allow you to monitor and control specific changes in database state. You can watch changes to rows and columns as they happen, and use an authorizer function to monitor and restrict what queries can do as they are compiled.

SQLite has support for threads as well, and offers a higher concurrency model for multi-threaded environments through the use of shared cache mode. This mode enables connections created by a single thread to share a common page cache, allowing the program to be more memory efficient. Furthermore, it is possible for multiple readers and one writer to work in the same database at the same time.

And believe it or not, you've only seen half of the API! Actually, you've seen more like three-quarters, but you are likely to find that what is in the next chapter—user-defined functions, aggregates, and collations—is every bit as useful and interesting as what is in this one.