CHAPTER 9

SQLite Internals

This chapter is a brief tour through SQLite's major subsystems. It was inspired from a SQLite presentation given by Richard Hipp at a convention I attended. Much of the material presented here was taken directly from the handouts. Even if you never look at SQLite's source code, you may find the material in this chapter quite interesting, as I did. While some of the material is apt to change with ongoing SQLite development, the major concepts presented here are not likely to change too dramatically.

By now, you should be familiar with the major components of SQLite's architecture. Chapter 1 provides a general description of each major component, and Chapter 5 elaborated a little on the B-tree and pager in the process of query execution and transaction management. Therefore, I will forgo any further introductions and dive right in. We will start with the virtual database engine, which is the heart of SQLite; proceed through the storage layer; and then finish with the compiler, which is perhaps the most complex part of the system.

The Virtual Database Engine

The VDBE is the center of the stack and is where execution takes place. To understand the system as a whole, it is easiest to start with the VDBE, as the modules above and below it essentially exist to serve it.

The VDBE is implemented in six source files: `vbde.c`, `vdbe.h`, `vdbeapi.c`, `vdbeInt.h`, and `vdbemem.c`. As explained in Chapter 5, a statement handle essentially represents a complete VDBE program to carry out a single SQL command. The statement holds everything needed to run the program, which includes the following:

- The VDBE program
- The program counter
- The names and data types of all result columns
- Bound parameter values
- The execution stack and a fixed number of numbered memory cells
- Other runtime state information, such as B-tree cursors

The VDBE is a virtual machine, and its program instructions resemble that of an assembly language. Each instruction in that language is composed of an *opcode* and three *operands*—P1, P2, and P3. The opcode, or operation code, it similar to a function. It performs a discrete task, and uses the operands as information needed to carry out that task. P1 is a 32-bit signed integer, P2 is a 31-bit unsigned integer, and P3 is a pointer to a string or structure. There are a total of 128 opcodes in SQLite at the time of this writing. The number and nature of opcodes are subject to change over time as they often adapt to support new features in SQLite. Unlike the C API, VDBE opcodes are always subject to change. Therefore, it is not safe for you to generate your own VDBE programs and try to run them across different versions of SQLite.

There are several C API functions that work directly with the VDBE:

- `sqlite3_bind_xxx()` functions
- `sqlite3_step()`
- `sqlite3_reset()`
- `sqlite3_column_xxx()` functions
- `sqlite3_finalize()`

Basically, all of the API calls used to execute a query and step through the results are associated with the VDBE. They all have one thing in common: they take a statement handle as an argument. This is because they all need either the VDBE code within the statement handle or its associated resources to carry out their job. Note that `sqlite3_prepare()` works with the front end (compiler) to produce the VDBE program. It has no part in execution.

The VDBE program for any SQL command can be obtained by prefixing the command with the `EXPLAIN` keyword. For example:

```
sqlite> .m col
sqlite> .h on
sqlite> .w 4 15 3 3 15
sqlite> explain select * from episodes;
addr  opcode           p1   p2   p3
----  ---------------  ---  ---  ---------------
0     Goto             0    12
1     Integer          0    0
2     OpenRead         0    2    # episodes
3     SetNumColumns    0    3
4     Rewind           0    10
5     Recno            0    0
6     Column           0    1
7     Column           0    2
8     Callback         3    0
9     Next             0    5
10    Close            0    0
11    Halt             0    0
12    Transaction      0    0
13    VerifyCookie     0    10
14    Goto             0    1
15    Noop             0    0
```

I included the first four commands in the listing for debugging and formatting. Also, I have compiled SQLite with the `SQLITE_DEBUG` option. This option provides more detailed information on the execution stack, such as including table names in the P3 operand as shown in the listing.

The Stack

A VDBE program is composed of different sections that accomplish specific tasks. In each section, there are usually a number of instructions that manipulate the stack, followed by an instruction that uses what is on the stack to carry out a task. Why is this so? Different instructions take different numbers of arguments. Some instructions only take one argument, and can use any of the operands to store it. Some instructions take no arguments. Some instructions take a variable number of arguments, in which case they cannot rely on just three operands to suffice.

In such cases, those instructions use the stack to supply their arguments. Thus, these instructions never operate by themselves, but with the help of other instructions that run before them, which load the stack with the needed arguments. In addition to the stack, the VDBE uses something called *memory cells* to hold intermediate values as well. This is useful for information that does not directly apply to the operation of subsequent instructions—such as record structures. Both stack entries and memory cells use the same underlying `Mem` data structure, defined in `vdbeInt.h` (*Int* is short for *internal*).

Program Body

For example, take the process of opening the `episodes` table for reading in the previous example. It is accomplished by a section of instructions, specifically instructions 1 through 3. The first instruction—`Integer`—prepares the stack for instruction 2. `OpenRead` takes the stack value and does something. Here's how it works.

As you know, SQLite can open multiple database files in the same connection using the `ATTACH` command. When SQLite opens a database, it assigns it an index. The main database is assigned index 0, the first attached database 1, and so on. The `Integer` instruction above `OpenRead` loads the database index on the stack. In this example, it loads 0 for the main database. The `OpenRead` instruction pulls this value off the stack and uses it to figure out which database to operate on. It uses the P2 operand to identify the location (the root page) of the table to open. It then opens a B-tree cursor on the given table within the given database. All of this is explained in the VDBE opcode documentation. For example, the `OpenRead` instruction is documented as follows:

> *Open a read-only cursor for the database table whose root page is P2 in a database file. The database file is determined by an integer from the top of the stack. 0 means the main database and 1 means the database used for temporary tables. Give the new cursor an identifier of P1. The P1 values need not be contiguous but all P1 values should be small integers. It is an error for P1 to be negative.*
>
> *If P2==0 then take the root page number from off of the stack.*
>
> *There will be a read lock on the database whenever there is an open cursor. If the database was unlocked prior to this instruction then a read lock is acquired as part of this instruction. A read lock allows other processes to read the database but prohibits any other process from modifying the database. The read lock is released when all cursors are closed. If this instruction attempts to get a read lock but fails, the script terminates with an SQLITE_BUSY error code.*

> *The P3 value is a pointer to a KeyInfo structure that defines the content and collating sequence of indices. P3 is NULL for cursors that are not pointing to indices.*

This documentation for `OpenRead` as well as all other instructions can be found directly in the source code, specifically in `vdbe.c`.

Finally, the `SetNumColumns` instruction sets the number of columns the cursor will hold, which corresponds to the number of columns in the table it points to. It uses operands P1 and P2. P1 specifies the cursor index (0 in this case, which is the index of the cursor just opened). P2 specifies the number of columns. The `episodes` table has three columns.

Continuing the example, the `Rewind` instruction places the cursor at the beginning of the table. It checks to ensure that the table is not empty (it has no records). If the table is empty, it will place the instruction pointer on the instruction specified in the P2 operand. In this case P2 is 10, so `Rewind` will jump to instruction 10 if the table is empty—in this example, the `Close` instruction (Figure 9-1).

Once `Rewind` has set the cursor position, the next section of instructions (instructions 5–9) takes over. This section's job is to iterate over all of the records in the table. `Recno` pushes the key value of the record onto the stack pointed to by the cursor identified in P1. The `Column` instruction pushes the value of the column with the ordinal given by P2 of the cursor given by P1. Together instructions 5, 6, and 7 place the `id` (primary key), `season`, and `name` fields onto the stack—all three fields of the `episodes` table—for the record that cursor 0 is pointing to. Notice that the P1 operand for all three instructions is 0. This refers to cursor 0. Next, the `Callback` instruction takes three values off the stack (specified by P1) and forms an array (or record) structure, which is stored in a memory cell. `Callback` will then suspend VDBE operation, yielding control back to `sqlite3_step()`, which will then return `SQLITE_ROW`.

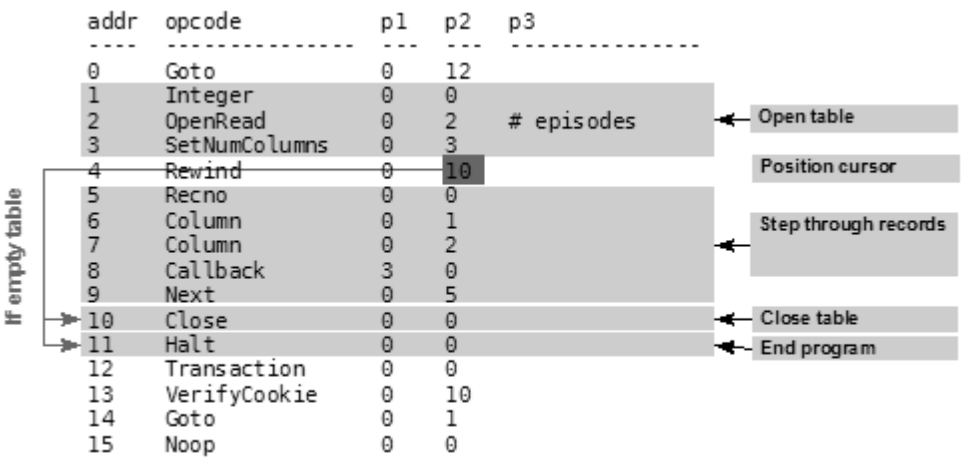

Figure 9-1. *VDBE steps: Open and Read*

Once the VDBE has created the record structure, it too is associated with the statement handle. The program can then use the `sqlite3_column_xxx()` functions to retrieve the fields from the record structure. On the next call to `sqlite3_step()`, the instruction pointer will be pointing to the `Next` instruction. The `Next` instruction advances the cursor to the next row in the

table. If there is another record, it jumps to the instruction given in P2, which in this case is instruction 5, which corresponds to the `Recno` instruction. Thus, it will go through another iteration, creating a new record structure, and `sqlite3_step()` will return `SQLITE_ROW`, indicating that there is another row to read. However, if there is not another row in the table, `Next` will fall through to the next instruction, which in this case is the `Close` instruction. `Close` will close the cursor and fall through to the `Halt` instruction, terminating the VDBE program, and `sqlite3_step()` will return `SQLITE_DONE`.

Program Startup and Shutdown

Now that we've seen the core of the program, let's look at the remaining sections, which relate to startup and initialization. These sections are shown in Figure 9-2. The first instruction is a `Goto`. `Goto` jumps to the instruction given in P2, which in this case is instruction 12.

Instruction 12 is a `Transaction` instruction, which starts a new transaction. It falls through to `VerifyCookie`, whose job is to ensure that the database schema has not changed since the VDBE program was compiled. This is an important concept in SQLite. Between the time SQL is compiled into VDBE code in `sqlite3_prepare()` and the time it is actually executed in `sqlite3_step()`, another SQL command could have run that changed the database schema (such as `ALTER TABLE`, `DROP TABLE`, or `CREATE TABLE`). When this happens, the schema version changes and it invalidates all statements that were compiled before the change. The current database schema is recorded at all times in the root page of the database file. Similarly, each statement object maintains a copy of the schema version it was compiled under for comparison. `VerifyCookie`'s job is to check that they match, and if they don't, to take appropriate action.

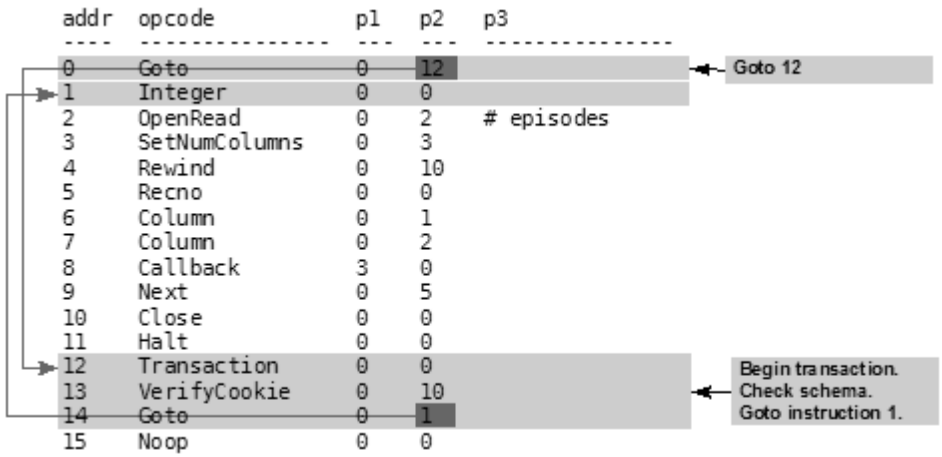

Figure 9-2. *VDBE steps: program startup*

`VerifyCookie` compares the database schema version on disk with the schema version given in P2—which corresponds to the statement's version referenced by the compiled code. If the schema has not changed since compile, the two will match. If they don't match, then the VDBE program (specifically the statement object that holds it) is no longer valid; it does not correspond to the most recent database schema. In this case, `VerifyCookie` will terminate the

VDBE program and arrange a SQLITE_SCHEMA error to be issued. In this case, the application needs to recompile the SQL statement, generating a new VDBE program based on the new schema. Recompiling could very well result in a different program. For example, what if someone dropped the episodes table and created a new table, which reused some of episodes' recycled pages? If the VDBE were to execute the invalid statement now, it would look for episodes (which no longer exists) starting at root page 2 (which has been recycled), and try to read records that no longer exist. The application would get junk for data.

However, when the schema versions do match, then the program counter steps from VerifyCookie to the following instruction—in this case Goto. Goto positions the program counter on instruction 1, which takes us to the main part of the program—opening the table and reading the records, as covered earlier. There are a few things to note here:

- The Transaction instruction doesn't acquire any locks in itself. It is the logical equivalent of a simple BEGIN. The actual shared lock is created by the OpenRead instruction. The lock is are freed when the transaction closes. This is determined by the Halt instruction, which cleans everything up. The action it takes depends on whether the connection is in autocommit mode or not.

- All of the storage space needed for the statement object (VDBE program) is determined before the start of the program. This is due primarily to two important facts. First, the depth of the stack is never greater than the number of instructions in the program. Second, the number of memory cells is never greater than the number of instructions (and is usually much less). Thus, even though SQL is dynamic in nature, SQLite can always compute how much memory a query will require and can allocate all of the resources before running the VDBE program.

Instruction Types

And this is how the VDBE works—one instruction at a time. Each instruction accomplishes a simple task and is often intertwined with instructions preceding or following it. If you look at the instruction set, you will see that opcodes fall into three general categories:

- **Value manipulation**: These instructions perform things like arithmetic operations (add, subtract, divide, etc.), logical operations (bitwise OR and AND), and string manipulation.

- **Data management**: These instructions manipulate data in memory and on disk. Memory instructions do things like stack manipulation and transfer data between stack entries and memory cells, and vice versa. Disk operations control the B-tree and pager modules—opening and manipulating cursors, beginning and ending transactions, and so forth.

- **Control flow**: Control instructions move the instruction pointer around both conditionally and unconditionally.

Once you become familiar with the instruction set, it is not hard to see what is going on. If nothing else, you can get a general feel for how the stack is used by some instructions to prepare the way for other instructions.

A Practical Example

Let's look at the example again, this time performing a VDBE trace on it. This will show us directly how the stack is being used between instructions. The first part of the program from startup to the first record is as follows:

```
sqlite> pragma vdbe_trace=on;
sqlite> select * from episodes;
VDBE Execution Trace:
SQL: [select * from episodes;]
   0 Goto             0   12
  12 Transaction      0    0
  13 VerifyCookie     0   10
  14 Goto             0    1
   1 Integer          0    0
Stack: i:0
   2 OpenRead         0    2 # episodes
   3 SetNumColumns    0    3
   4 Rewind           0   10
   5 Recno            0    0
Stack: i:0
   6 Column           0    1
Stack: NULL i:0
   7 Column           0    2
Stack:  s18[Good News Bad N](8) NULL i:0
   8 Callback         3    0
0||Good News Bad News
   9 Next             0    5
```

You can see the startup in the first four steps, followed by the `Integer` command, which specifies on the stack the database index upon which to open the cursor. `OpenRead` will pop it from the stack and open a cursor on the table with root page 2 (given by P2). Step down into the loop part and you see `Recno` and `Column` instructions pushing field values of the current record onto the stack. `Callback` then pops all of them off the stack and uses them to form a record structure, which is then stored in the statement handle. `sqlite3_step()` returns a `SQLITE_ROW`, indicating that a row is available for reading. The application accesses the data with `sqlite3_column_xxx()`, and proceeds to the next record by calling `sqlite3_step()` again. `sqlite3_step()` then resumes VDBE execution at the `Next` instruction, which moves on to the next record. And so on and so forth.

So not only can you see the stack at work, but you can also get some idea as to how the SQLite C API functions are intertwined within VDBE code. From here on out, it is a matter of just playing with the VDBE to see what the instructions do. For instance:

```
sqlite> explain BEGIN;

addr  opcode           p1   p2   p3
----  ---------------  ---  ---  ---------------
0     AutoCommit       0    0
1     Halt             0    0
2     Noop             0    0
sqlite> explain COMMIT;

addr  opcode           p1   p2   p3
----  ---------------  ---  ---  ---------------
0     AutoCommit       1    0
1     Halt             0    0
2     Noop             0    0
sqlite> explain ROLLBACK;

addr  opcode           p1   p2   p3
----  ---------------  ---  ---  ---------------
0     AutoCommit       1    1
1     Halt             0    0
2     Noop             0    0
```

You can see here that transactions work by toggling the `AutoCommit` instruction. This instruction must therefore be related to initiating the transfer of modified pages to disk. Try different things and see what happens. After a little practice, you may very well be able to recognize enough of the instruction patterns to use `EXPLAIN` to optimize queries.

The B-Tree and Pager Modules

The B-tree provides the VDBE with *O(logN)* lookup, insert, and delete as well as *O(1)* bidirectional traversal of records. It is self-balancing, and automatically manages both defragmentation and space reclamation. The B-tree itself has no notion of reading or writing to disk. It only concerns itself with the relationship between pages. The B-tree notifies the pager when it needs a page, and also notifies it when it is about to modify a page. When modifying a page, the pager ensures that the original page is first copied out to the journal file. Similarly, the B-tree notifies the pager when it is done writing, and the pager determines what needs to be done based on the transaction state.

Database File Format

All pages in a database are numbered sequentially, beginning with 1. A database is composed of multiple B-trees—one B-tree for each table and index (B+trees are used for tables, B-trees for indexes). Each table or index in a database has a root page that defines the location of its first page. The root pages for all indexes and tables are stored in the `sqlite_master` table.

The first page in the database—Page 1—is special. The first 100 bytes of Page 1 contain a special header (the file header) that describes the database file. It includes such information as the library version, schema version, page size, encoding, whether or not autovacuum is enabled—all of the permanent database settings you configure when creating a database along with any other parameters that have been set by various pragmas. The exact contents of the header are documented in `btree.c`. Page 1 is also the root page of the `sqlite_master` table.

Page Reuse and Vacuum

SQLite recycles pages using a free list. That is, when all of the records are deleted out of a page, the page is placed on a list reserved for reuse. When new information is later added, nearby pages are first selected before any new pages are created (expanding the database file). Running a `VACUUM` command purges the free list and thereby shrinks the database. In actuality, it rebuilds the database in a new file so that all in-use pages are copied over, while free-list pages are not. The end result is a new, compacted database. When autovacuum is enabled in a database, SQLite doesn't use a free list, and automatically shrinks the database upon each commit.

B-Tree Records

Pages in a B-tree are made up of B-tree records, also called payloads. They are not database records in the sense you might think—formatted with the columns in a table. They are more primitive than that. A B-tree record, or payload, has only two fields: a key field and a data field. The key field is the `ROWID` value, or primary key value that is present for every table in the database. The data field, from the B-tree perspective, is an amorphous blob that can contain anything. Ultimately, the database records are stored inside the data fields. The B-tree's job is order and navigation, and it primarily needs only the key field to do its work (although there is one exception in B+trees, which is addressed next). Furthermore, payloads are variable size, as are their internal key and data fields. On average, each page usually holds multiple payloads; however, it is possible for a payload to span multiple pages if it is too large to fit on one page (e.g., records containing `BLOB`s).

B+Trees

B-tree records are stored in key order. All keys must be unique within a single B-tree (this is automatically guaranteed as the keys correspond to the `ROWID` primary key, and SQLite takes care of that field for you). Tables use B+trees, which do not contain table data (database records) in the internal pages. An example B+tree representation of a table is shown in Figure 9-3.

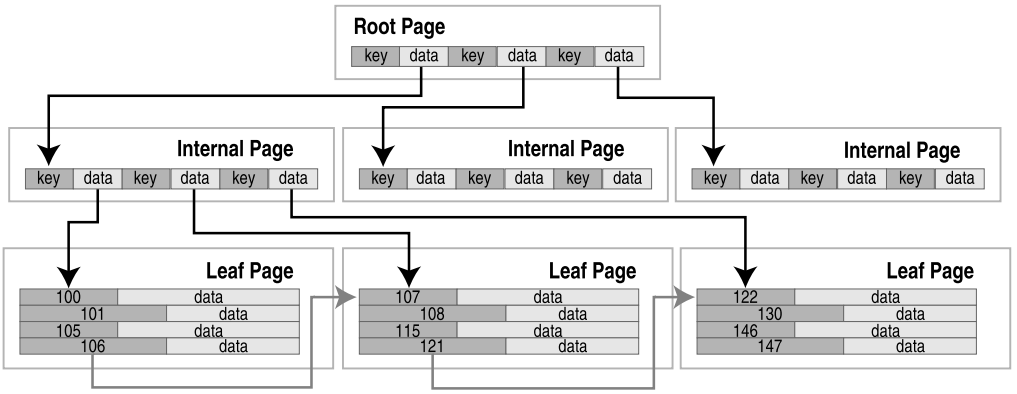

Figure 9-3. *B+tree organization (tables)*

The root page and internal pages in B+trees are all about navigation. The data fields in these pages are all pointers to the pages below them—they contain keys only. All database records are stored on the leaf pages. On the leaf level, records and pages are arranged in key order so it is possible for B-tree cursors to traverse records (horizontally), both forward and backward, using only the leaf pages. This is what makes traversal possible in *O(1)* time.

Records and Fields

The database records in the data fields of the leaf pages are managed by the VDBE (as in the `Callback` instruction described earlier). The database record is stored in binary form using a specialized record format that describes all of the fields in the record. The record format consists of a contiguous stream of bytes organized into a logical header and a data segment. The header segment includes the header size (represented as a variable-sized 64-bit integer value) and an array of types (also variable 64-bit integers), which describe each field stored in the data segment, as shown in Figure 9-4. Variable 64-bit integers are implemented using a Huffman code.

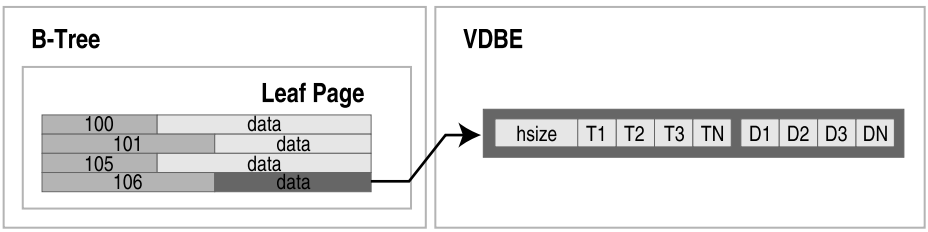

Figure 9-4. *Record structure*

The number of type entries corresponds to the number of fields in the record. Furthermore, each index in the type array corresponds to the same index in the field array. A type entry specifies the data type and size of its corresponding field value. The possible types and their meanings are listed in Table 9-1.

Table 9-1. *Field Type Values*

Type Value	Meaning	Length of Data
0	NULL	0
N in 1..4	Signed integer	N
5	Signed integer	6
6	Signed integer	8
7	IEEE float	8
8-11	Reserved for future use	N/A
N>12 and even	BLOB	(N-12)/2
N>13 and odd	TEXT	(N-13)/2

For example, take the first record in the episodes table:

```
sqlite> SELECT * FROM episodes ORDER BY id LIMIT 1;
id   season  name
---  ------  --------------------
0    1       Good News Bad News
```

The internal record format for this record is shown in Figure 9-5.

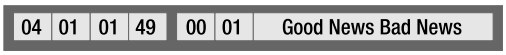

Figure 9-5. *First record in the episodes table*

The header is 4 bytes long. The header size reflects this and itself is encoded as a single byte. The first type, corresponding to the id field, is a 1-byte signed integer. The second type, corresponding to the season field, is as well. The name type entry is an odd number, meaning it is a text value. Its size is therefore given by (49-13)/2=17 bytes. With the information, the VDBE can parse the data segment of the record and extract the individual fields.

Hierarchical Data Organization

Basically, each module in the stack deals with a specific unit of data. From the bottom up, data becomes more refined and more specific. From the top down, it becomes more aggregated and amorphous. Specifically, the C-API deals in field values, the VDBE in records, the B-tree in keys and data, the pager in pages, the OS interface in binary data and raw storage. This is illustrated in Figure 9-6.

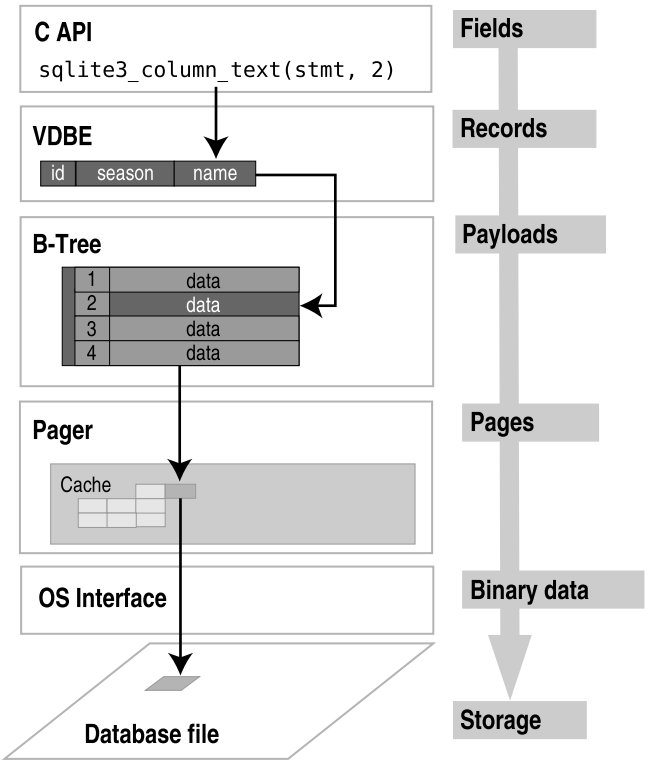

Figure 9-6. *Modules and their associated data*

Each module takes part in managing its own specific portion of the data in the database, and relies on the layer below it to supply it with a more crude form from which to extract its respective pieces.

Overflow Pages

As mentioned earlier, payloads and their contents can have variable sizes. However, pages are fixed in size. Therefore, there is always the possibility that a given payload could be too large to fit in a single page. When this happens, the excess payload is spilled out onto a linked list of overflow pages. From this point on, the payload takes on the form of a linked list of sorts, as shown in Figure 9-7.

The fourth payload in the figure is too large to fit on the page. As a result, the B-tree module creates an overflow page to accommodate. It turns out that one page won't suffice, so it links a second. This is essentially the way binary large objects are handled. One thing to keep in mind when you are using really large `BLOB`s is that they are ultimately being stored as a linked list of pages. If the `BLOB` is large enough, this can become inefficient, in which case you might consider dedicating an external file for the `BLOB` and keeping this filename in the record instead.

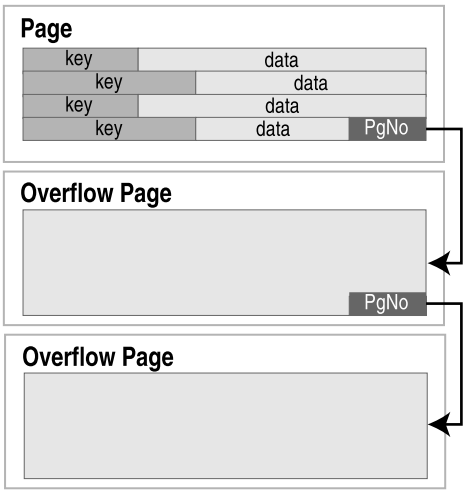

Figure 9-7. *Overflow pages*

The B-Tree API

The B-tree module has its own API, which is separate from and can be used independently of the C API. That is, you could use it as a standalone B-tree library, or access tables in a SQLite database directory, if you wish. An added benefit of SQLite's B-tree module is that it includes native support for transactions. Everything you know about the transactions, locks, and journal file are handled by the pager, which serves the B-tree module. The API can be grouped into functions according to general purpose.

Access and Transaction Functions

Database and transaction routines include the following:

- `sqlite3BtreeOpen`: Opens a new database file. Returns a B-tree object.
- `sqlite3BtreeClose`: Closes a database.
- `sqlite3BtreeBeginTrans`: Starts a new transaction.
- `sqlite3BtreeCommit`: Commits the current transaction.
- `sqlite3BtreeRollback`: Rolls back the current transaction.
- `sqlite3BtreeBeginStmt`: Starts a statement transaction.
- `sqlite3BtreeCommitStmt`: Commits a statement transaction.
- `sqlite3BtreeRollbackStmt`: Rolls back a statement transaction.

Table Functions

Table management routines include the following:

- `sqlite3BtreeCreateTable`: Creates a new, empty B-tree in a database file. Flags in the argument determine whether to use a table format (B+tree) or index format (B-tree).
- `sqlite3BtreeDropTable`: Destroys a B-tree in a database file.
- `sqlite3BtreeClearTable`: Removes all data from a B-tree, but keeps the B-tree intact.

Cursor Functions

Cursor functions include the following:

- `sqlite3BtreeCursor`: Creates a new cursor pointing to a particular B-tree. Cursors can be either a read cursor or a write cursor. Read and write cursors may not exist in the same B-tree at the same time.
- `sqlite3BtreeCloseCursor`: Closes the B-tree cursor.
- `sqlite3BtreeFirst`: Moves the cursor to the first element in a B-tree.
- `sqlite3BtreeLast`: Moves the cursor to the last element in a B-tree.
- `sqlite3BtreeNext`: Moves the cursor to the next element after the one it is currently pointing to.
- `sqlite3BtreePrevious`: Moves the cursor to the previous element before the one it is currently pointing to.
- `sqlite3BtreeMoveto`: Moves the cursor to an element that matches the key value passed in as a parameter. If there is no match, leaves the cursor pointing to an element that would be on either side of the matching element, had it existed.

Record Functions

Key and record functions include the following:

- `sqlite3BtreeDelete`: Deletes the record that the cursor is pointing to.
- `sqlite3BtreeInsert`: Inserts a new element in the appropriate place of the B-tree.
- `sqlite3BtreeKeySize`: Returns the number of bytes in the key of the record that the cursor is pointing to.
- `sqlite3BtreeKey`: Returns the key of the record the cursor is currently pointing to.
- `sqlite3BtreeDataSize`: Returns the number of bytes in the data record that the cursor is currently pointing to.
- `sqlite3BtreeData`: Returns the data in the record the cursor is currently pointing to.

Configuration Functions

Functions to set various parameters include the following:

- `sqlite3BtreeSetCacheSize`: Controls the page cache size as well as the synchronous writes (as defined in the `synchronous` pragma).
- `sqlite3BtreeSetSafetyLevel`: Changes the way data is synced to disk in order to increase or decrease how well the database resists damage due to OS crashes and power failures. Level 1 is the same as asynchronous (no `syncs()` occur and there is a high probability of damage). This is the equivalent to pragma `synchronous=OFF`. Level 2 is the default. There is a very low but non-zero probability of damage. This is the equivalent to pragma `synchronous=NORMAL`. Level 3 reduces the probability of damage to near zero but with a write performance reduction. This is the equivalent to pragma `synchronous=FULL`.
- `sqlite3BtreeSetPageSize`: Sets the database page size.
- `sqlite3BtreeGetPageSize`: Returns the database page size.
- `sqlite3BtreeSetAutoVacuum`: Sets the autovacuum property of the database.
- `sqlite3BtreeGetAutoVacuum`: Returns whether the database uses autovacuum.
- `sqlite3BtreeSetBusyHandler`: Sets the busy handler.

There are more functions, all of which are very well documented in `btree.h` and `btree.c`, but those listed here give you some idea of the parts of the API that are implemented in the B-tree layer, as well as what this layer can do in its own right.

The Compiler

So far, you have seen how things operate from the VDBE down the stack through the storage and OS layers. Now let's look at how the VDBE program is constructed to begin with. This is the job of the front end. The front end takes a single SQL command as an input and ends with a complete, optimized VDBE program. This happens in three stages: tokenizing, parsing, and code generation.

The Tokenizer

The first step of compilation is tokenizing the SQL command. The tokenizer splits a command into a stream of individual tokens. A token is simply a character or sequence of characters that has a specific meaning (in this case some meaning within SQL). Each token has an associated token class, which is simply a numeric identifier describing what the token is. For example, a left parenthesis token is classified as `TK_LP`, and the `SELECT` keyword is classified as `TK_SELECT`. All token classes are defined in `parse.h`. For instance, the SQL statement

```
SELECT rowid FROM foo where name='bar' LIMIT 1 ORDER BY rowid;
```

would be processed by the tokenizer as shown in part in Table 9-2.

In short, the tokenizer breaks up each unit of text, classifies it, and sends it to the parser (discarding white space).

Table 9-2. *A Tokenized SELECT statement*

Text	Token Class	Action
SELECT	TK_SELECT	Sent to parser
" "	TK_SPACE	Discarded
Rowid	TK_ID	Sent to parser
" "	TK_SPACE	Discarded
FROM	TK_FROM	Sent to parser
" "	TK_SPACE	Discarded
foo	TK_ID	Sent to parser
""	TK_SPACE	Discarded
WHERE	TK_WHERE	Sent to parser
" "	TK_SPACE	Discarded
name	TK_ID	Sent to parser
=	TK_EQ	Sent to parser

Keywords

The tokenizer is hand-coded and located in `tokenize.c`. While it is hand-coded, it does use generated code to classify SQL keywords, which are defined in `keywordhash.h`. This file is an optimization that compacts all SQL keywords into the smallest possible buffer by overlapping common character sequences in keywords. SQLite identifies keyword entries using arrays that define the offsets and size of each keyword. The method is a space optimization intended to help embedded applications. An example of the generated buffer is as follows:

```
static int keywordCode(const char *z, int n){
  static const char zText[537] =
    "ABORTABLEFTEMPORARYADDDATABASELECTHENDEFAULTRANSACTIONATURALTER"
    "AISEACHECKEYAFTEREFERENCESCAPELSEXCEPTRIGGEREGEXPLAINITIALLYANALYZE"
    "XCLUSIVEXISTSTATEMENTANDEFERRABLEATTACHAVINGLOBEFOREIGNOREINDEX"
    "AUTOINCREMENTBEGINNERENAMEBETWEENOTNULLIKEBYCASCADEFERREDELETE"
    "CASECASTCOLLATECOLUMNCOMMITCONFLICTCONSTRAINTERSECTCREATECROSS"
    "CURRENT_DATECURRENT_TIMESTAMPLANDESCDETACHDISTINCTDROPRAGMATCH"
    "FAILIMITFROMFULLGROUPDATEIFIMMEDIATEINSERTINSTEADINTOFFSETISNULL"
    "JOINORDEREPLACEOUTERESTRICTPRIMARYQUERYRIGHTROLLBACKROWHENUNION"
    "UNIQUEUSINGVACUUMVALUESVIEWHERE";
```

The `keywordhash.h` file includes a routine `sqlite3KeywordCode()`, which allows the tokenizer to quickly match the keyword with its appropriate token class with minimal space. So, the tokenizer first tries to match a token with what it knows, and failing that, it resorts to `sqlite3KeywordCode()`, which will return either a keyword token class or a generic `TK_ID`.

The tokenizer and parser work hand in hand, one token at a time. As the tokenizer resolves each token, it passes the token to the parser. The parser takes the tokens and builds a parse tree, which is a hierarchical representation of the statement.

The Parser

The parser is generated from SQLite's custom parser generator—the Lemon parser generator. The parser uses grammar rules defined in `parse.y` to organize the tokens into a parse tree. The theory behind the parser is beyond the scope of this chapter—there are many textbooks that cover the topic. The point is that the parser converts a stream of tokens into a hierarchical data structure, called a parse tree, which represents the SQL command.

The parse tree is primarily composed of expressions and lists of expressions. An expression itself is a recursive structure that can contain subexpressions under it. For example, the `WHERE` clause in a `SELECT` parse tree is represented by a single expression. The `SELECT` clause, on the other hand, is represented as a list of expressions; each expression is a column that will be returned in the result set. For example, a very simplified representation of the statement

```
SELECT rowid, name, season FROM episodes WHERE rowid=1 LIMIT 1
```

might be arranged into a parse tree as shown in Figure 9-8.

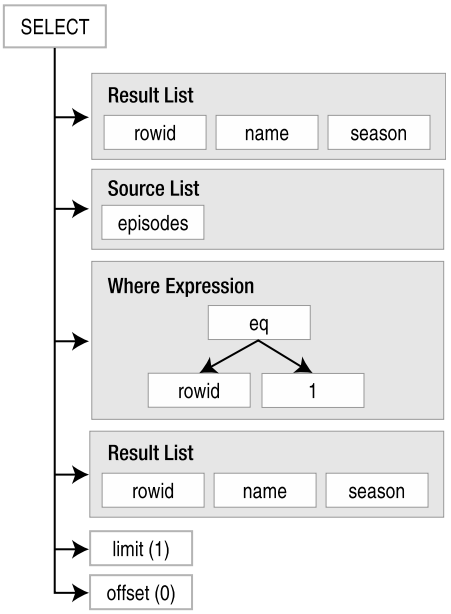

Figure 9-8. *Simplified parse tree representation*

Once the statement has been completely tokenized and parsed, the parse tree is passed to the code generator.

The Code Generator

The code generator is the largest and most complex part of SQLite. Unlike most other modules, the code generator does not have a well-defined interface, but is rather closely tied to the parser. It is made up of many source files, most of which are specific to SQL operations. For example, the code to generate `SELECT` statements is kept in `select.c`. Other source files include `update.c`, `insert.c`, `delete.c`, `trigger.c`, and so on.

The code generator takes the parse tree and produces the VDBE program to execute the statement. Each part of the tree more or less is handled by a specific routine that generates a sequence of VDBE instructions to accomplish a specific task. The values for the operands are taken from the data structures associated with the parse tree. For example, the function to open a table for reading is implemented as follows:

```
/*
** Generate code that will open a table for reading.
*/
void sqlite3OpenTableForReading(
  Vdbe *v,        /* Generate code into this VDBE */
  int iCur,       /* The cursor number of the table */
  Table *pTab     /* The table to be opened */
){
  sqlite3VdbeAddOp(v, OP_Integer, pTab->iDb, 0);
  sqlite3VdbeAddOp(v, OP_OpenRead, iCur, pTab->tnum);
  VdbeComment((v, "# %s", pTab->zName));
  sqlite3VdbeAddOp(v, OP_SetNumColumns, iCur, pTab->nCol);
}
```

The `sqlite3vdbeAddOp` function takes three arguments: a VDBE instance (to which it will add instructions), an instruction (or opcode), and two operands. It adds exactly one instruction to the VDBE instance's program. In this case, `sqlite3OpenTableForReading` adds the three instructions (1–3) to open a B-tree table for reading, just as it was shown in Figure 9-1, which are listed here again for convenience:

```
sqlite> explain select * from episodes;
addr  opcode           p1   p2   p3
----  ---------------  ---  ---  ---------------
0     Goto             0    12
1     Integer          0    0
2     OpenRead         0    2    # episodes
3     SetNumColumns    0    3
4     Rewind           0    10
5     Recno            0    0
6     Column           0    1
7     Column           0    2
8     Callback         3    0
9     Next             0    5
10    Close            0    0
11    Halt             0    0
12    Transaction      0    0
```

```
13    VerifyCookie     0    10
14    Goto             0    1
15    Noop             0    0
```

For example, the second line of `sqlite3OpenTableForReading` is

```
sqlite3VdbeAddOp(v, OP_OpenRead, iCur, pTab->tnum);
```

This creates instruction 3 in the program. The P2 value refers to the root page of the table to be opened. This is provided by the table data structure pointed to by `pTab`, specifically the `tnum` member, which holds the table's root page number. So whenever a parse tree calls for a table to be opened, it calls `sqlite3OpenTableForReading` to add the corresponding VDBE instructions to do so.

In short, code generation is done by many different routines across many different files in the code. The routines analyze a specific portion of the parse tree and create corresponding VDBE instructions to accomplish a specific task. The code generator contains a wide variety of routines for generating instructions. General functions are available that are used in many places; `sqlite3ExprCode`, for example, takes an expression and generates the instructions needed to evaluate it. Likewise, there are highly specific routines that accomplish unique tasks, such as `sqlite3FinishTrigger`, which generates the VDBE instructions to complete the process of building a trigger.

The Optimizer

The code generator does more than just code generation, however. It performs query optimization as well. The optimizer is part of the code generator, and lives in `where.c`. Code that creates the `WHERE` clause is shared by other modules, such as `select.c`, `update.c`, and `delete.c`. Each of these modules calls `sqlite3WhereBegin()` to generate the `WHERE` clause code to produce the search instructions, and then add their own VDBE code after it to process the rows that are returned. They then conclude that code with `sqlite3WhereEnd()`, which adds the VDBE instructions that concludes the `WHERE` clause code. The general structure is shown in Figure 9-9.

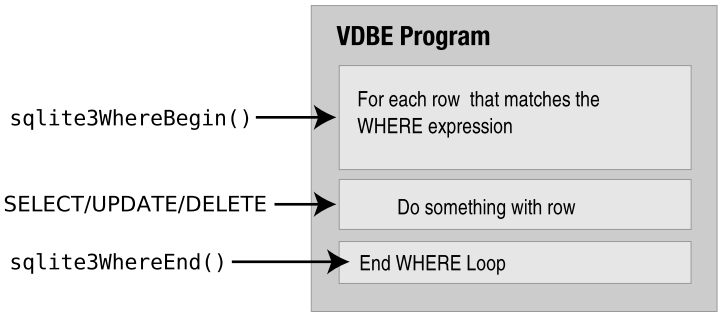

Figure 9-9. *WHERE clause VDBE code generation*

Optimizations take place in `sqlite3WhereBegin()`. Based on what is being done, it looks for indexes that might be used, expressions that can be rewritten, and so forth.

To get a feel for how this works, let's start with a simple `SELECT` with no `WHERE` clause, as shown in Figure 9-10.

```
sqlite> explain SELECT name FROM episodes;
addr  opcode         p1   p2   p3
----  -------------  ---  ---  ---------------
0     Goto           0    10
1     Integer        0    0
2     OpenRead       0    2    # episodes
3     SetNumColumns  0    3
4     Rewind         0    8                    sqlite3WhereBegin()
5     Column         0    2
6     Callback       1    0                    SELECT loop
7     Next           0    5
8     Close          0    0                    sqlite3WhereEnd()
9     Halt           0    0
10    Transaction    0    0
11    VerifyCookie   0    14
12    Goto           0    1
13    Noop           0    0
```

Figure 9-10. *A SELECT statement with no WHERE clause*

By now, you can start to recognize the various code generation sections. Instructions 1–3 open `episodes` for reading, and instructions 4–9 iterate through all records, selecting the `name` column (identified here by the `Column` instruction). Even though there was no `WHERE` clause here, `sqlite3WhereBegin()` and `sqlite3WhereEnd()` are called and generate instructions. Here, the query calls for a table scan. `sqlite3WhereBegin()` adds the `Rewind` instruction, which places the cursor at the beginning of the table. Following that, instructions 5–6 are from the select code, and `sqlite3WhereEnd()` concludes with instructions 7–8. Instructions 0 and 10–13 are the standard prologue instructions for executing a command.

Next, let's look at what a `WHERE` clause does (Figure 9-11).

```
sqlite> explain SELECT name FROM episodes WHERE name='Soup Nazi';
addr  opcode         p1   p2   p3
----  -------------  ---  ---  ---------------
0     Goto           0    13
1     Integer        0    0
2     OpenRead       0    2    # episodes
3     SetNumColumns  0    3
4     Rewind         0    11
5     Column         0    2
6     String8        0    0    Soup Nazi       sqlite3WhereBegin()
7     Ne             296  10   collseq(BINARY)
8     Column         0    2
9     Callback       1    0
10    Next           0    5                    sqlite3WhereEnd()
11    Close          0    0
12    Halt           0    0
13    Transaction    0    0
14    VerifyCookie   0    15
15    Goto           0    1
16    Noop           0    0
```

Figure 9-11. *Instructions relating to a WHERE clause*

Instructions 4–6, produced by `sqlite3WhereBegin()`, add the constraint `name='Soup Nazi'`. Everything else is unchanged. In both cases the `WHERE` clause performs a full table scan, as is evidenced by the `Next` instruction in the `WhereEnd` segment.

Now, create an index on the `name` column and see what happens (Figure 9-12).

Now, `sqlite3WhereBegin()` sees that there is an index it can use and writes in VDBE instructions to do so. This is essentially how the optimizer works: as the `WHERE` component of the code generator.

```
sqlite> create index ep_name_idx on episodes (name);
sqlite> explain SELECT name FROM episodes WHERE name='Soup Nazi';
addr  opcode           p1   p2   p3
----  ---------------  ---  ---  -----------------
0     Goto             0    21
1     Integer          0    0
2     OpenRead         1    33   keyinfo(1,BINARY)
3     KeyAsData        1    1
4     SetNumColumns    1    2
5     String8          0    0    Soup Nazi
6     NotNull          -1   9
7     Pop              1    0
8     Goto             0    19                        <- sqlite3WhereBegin()
9     MakeRecord       1    0    t
10    MemStore         0    0
11    MoveGe           1    19
12    MemLoad          0    0
13    IdxGE            1    19   +
14    RowKey           1    0
15    IdxIsNull        1    18
16    Column           1    0
17    Callback         1    0
18    Next             1    12
19    Close            1    0                         <- sqlite3WhereEnd()
20    Halt             0    0
21    Transaction      0    0
22    VerifyCookie     0    16
23    Goto             0    1
24    Noop             0    0
```

Figure 9-12. *Instructions relating to index use in the WHERE clause*

Summary

And that concludes our journey through SQLite, not only in this chapter, but the book as well. I hope you have enjoyed it. As you've seen in Chapter 1, SQLite is more than merely a free database. It is a well-written software library with a wide range of applications. It is a database, a utility, and a helpful programming tool.

What you have seen in this chapter barely scratches the surface, but it should give you a better idea about how things work nonetheless. And it also gives you an appreciation for how elegantly SQLite approaches a very complex problem. You know firsthand how big SQL is, and you've seen the complexity of the relational model behind it. Yet SQLite is a small library and manages to put many of these concepts to work in a small amount of code. It does this by breaking query processing down into discrete tasks. Each of its modules accomplishes a specific task

and works closely with the modules above and below it. This design keeps things simple, manageable, and reliable. Add to this a large and thorough testing suite and that is how SQLite packs a lot of relational power into a very small package.

From a programming standpoint, SQLite is not only useful in its own right, but there is a lot you can learn from it as well. Even though it is small, it employs quite a variety of different concepts: tokenizing, parsing, virtual machines, B-trees and B+trees, caching, locking, transactions, memory management, and more. Not only does it implement all of these things, it does so while also satisfying tough design constraints: portability, flexibility, simplicity, efficiency, and reliability. Therefore, by looking at SQLite's implementation, you can study not only a particular concept but also how it is implemented from this perspective. And if a particular implementation works for you, the code is yours for the taking.