The Virtual Database Engine of SQLite

> **Obsolete Documentation Warning:** This document describes the virtual machine used in SQLite version 2.8.0. The virtual machine in SQLite version 3.0 and 3.1 is similar in concept but is now register-based instead of stack-based, has five operands per opcode instead of three, and has a different set of opcodes from those shown below. See the virtual machine instructions document for the current set of VDBE opcodes and a brief overview of how the VDBE operates. This document is retained as an historical reference.

If you want to know how the SQLite library works internally, you need to begin with a solid understanding of the Virtual Database Engine or VDBE. The VDBE occurs right in the middle of the processing stream (see the architecture diagram) and so it seems to touch most parts of the library. Even parts of the code that do not directly interact with the VDBE are usually in a supporting role. The VDBE really is the heart of SQLite.

This article is a brief introduction to how the VDBE works and in particular how the various VDBE instructions (documented here) work together to do useful things with the database. The style is tutorial, beginning with simple tasks and working toward solving more complex problems. Along the way we will visit most submodules in the SQLite library. After completing this tutorial, you should have a pretty good understanding of how SQLite works and will be ready to begin studying the actual source code.

Preliminaries

The VDBE implements a virtual computer that runs a program in its virtual machine language. The goal of each program is to interrogate or change the database. Toward this end, the machine language that the VDBE implements is specifically designed to search, read, and modify databases.

Each instruction of the VDBE language contains an opcode and three operands labeled P1, P2, and P3. Operand P1 is an arbitrary integer. P2 is a non-negative integer. P3 is a pointer to a data structure or zero-terminated string, possibly null. Only a few VDBE instructions use all three operands. Many instructions use only one or two operands. A significant number of instructions use no operands at all but instead take their data and store their results on the execution stack. The details of

what each instruction does and which operands it uses are described in the separate opcode description document.

A VDBE program begins execution on instruction 0 and continues with successive instructions until it either (1) encounters a fatal error, (2) executes a Halt instruction, or (3) advances the program counter past the last instruction of the program. When the VDBE completes execution, all open database cursors are closed, all memory is freed, and everything is popped from the stack. So there are never any worries about memory leaks or undeallocated resources.

If you have done any assembly language programming or have worked with any kind of abstract machine before, all of these details should be familiar to you. So let's jump right in and start looking as some code.

Inserting Records Into The Database

We begin with a problem that can be solved using a VDBE program that is only a few instructions long. Suppose we have an SQL table that was created like this:

```
CREATE TABLE examp(one text, two int);
```

In words, we have a database table named "examp" that has two columns of data named "one" and "two". Now suppose we want to insert a single record into this table. Like this:

```
INSERT INTO examp VALUES('Hello, World!',99);
```

We can see the VDBE program that SQLite uses to implement this INSERT using the **sqlite** command-line utility. First start up **sqlite** on a new, empty database, then create the table. Next change the output format of **sqlite** to a form that is designed to work with VDBE program dumps by entering the ".explain" command. Finally, enter the [INSERT] statement shown above, but precede the [INSERT] with the special keyword [EXPLAIN]. The [EXPLAIN] keyword will cause **sqlite** to print the VDBE program rather than execute it. We have:

```
$ sqlite test_database_1
sqlite> CREATE TABLE examp(one text, two int);
sqlite> .explain
sqlite> EXPLAIN INSERT INTO examp VALUES('Hello, World!',99);
addr  opcode        p1     p2     p3
----  ------------  -----  -----  ----------------------------------
0     Transaction   0      0
1     VerifyCookie  0      81
2     Transaction   1      0
3     Integer       0      0
4     OpenWrite     0      3      examp
5     NewRecno      0      0
6     String        0      0      Hello, World!
```

```
7     Integer       99     0      99
8     MakeRecord    2      0
9     PutIntKey     0      1
10    Close         0      0
11    Commit        0      0
12    Halt          0      0
```

As you can see above, our simple insert statement is implemented in 12 instructions. The first 3 and last 2 instructions are a standard prologue and epilogue, so the real work is done in the middle 7 instructions. There are no jumps, so the program executes once through from top to bottom. Let's now look at each instruction in detail.

```
0     Transaction   0      0
1     VerifyCookie  0      81
2     Transaction   1      0
```

The instruction Transaction begins a transaction. The transaction ends when a Commit or Rollback opcode is encountered. P1 is the index of the database file on which the transaction is started. Index 0 is the main database file. A write lock is obtained on the database file when a transaction is started. No other process can read or write the file while the transaction is underway. Starting a transaction also creates a rollback journal. A transaction must be started before any changes can be made to the database.

The instruction VerifyCookie checks cookie 0 (the database schema version) to make sure it is equal to P2 (the value obtained when the database schema was last read). P1 is the database number (0 for the main database). This is done to make sure the database schema hasn't been changed by another thread, in which case it has to be reread.

The second Transaction instruction begins a transaction and starts a rollback journal for database 1, the database used for temporary tables.

```
3     Integer       0      0
4     OpenWrite     0      3      examp
```

The instruction Integer pushes the integer value P1 (0) onto the stack. Here 0 is the number of the database to use in the following OpenWrite instruction. If P3 is not NULL then it is a string representation of the same integer. Afterwards the stack looks like this:

```
(integer) 0
```

The instruction OpenWrite opens a new read/write cursor with handle P1 (0 in this case) on table "examp", whose root page is P2 (3, in this database file). Cursor handles can be any non-negative integer. But the VDBE allocates cursors in an array with the size of the array being one more than the largest cursor. So to conserve

memory, it is best to use handles beginning with zero and working upward consecutively. Here P3 ("examp") is the name of the table being opened, but this is unused, and only generated to make the code easier to read. This instruction pops the database number to use (0, the main database) from the top of the stack, so afterwards the stack is empty again.

```
5     NewRecno      0     0
```

The instruction NewRecno creates a new integer record number for the table pointed to by cursor P1. The record number is one not currently used as a key in the table. The new record number is pushed onto the stack. Afterwards the stack looks like this:

(integer) new record key

```
6     String        0     0     Hello, World!
```

The instruction String pushes its P3 operand onto the stack. Afterwards the stack looks like this:

(string) "Hello, World!"
(integer) new record key

```
7     Integer       99    0     99
```

The instruction Integer pushes its P1 operand (99) onto the stack. Afterwards the stack looks like this:

(integer) 99
(string) "Hello, World!"
(integer) new record key

```
8     MakeRecord    2     0
```

The instruction MakeRecord pops the top P1 elements off the stack (2 in this case) and converts them into the binary format used for storing records in a database file. (See the file format description for details.) The new record generated by the MakeRecord instruction is pushed back onto the stack. Afterwards the stack looks like this:

(record) "Hello, World!", 99
(integer) new record key

```
9     PutIntKey     0     1
```

The instruction PutIntKey uses the top 2 stack entries to write an entry into the table pointed to by cursor P1. A new entry is created if it doesn't already exist or the

data for an existing entry is overwritten. The record data is the top stack entry, and the key is the next entry down. The stack is popped twice by this instruction. Because operand P2 is 1 the row change count is incremented and the rowid is stored for subsequent return by the sqlite_last_insert_rowid() function. If P2 is 0 the row change count is unmodified. This instruction is where the insert actually occurs.

```
10    Close          0      0
```

The instruction Close closes a cursor previously opened as P1 (0, the only open cursor). If P1 is not currently open, this instruction is a no-op.

```
11    Commit         0      0
```

The instruction Commit causes all modifications to the database that have been made since the last Transaction to actually take effect. No additional modifications are allowed until another transaction is started. The Commit instruction deletes the journal file and releases the write lock on the database. A read lock continues to be held if there are still cursors open.

```
12    Halt           0      0
```

The instruction Halt causes the VDBE engine to exit immediately. All open cursors, Lists, Sorts, etc are closed automatically. P1 is the result code returned by sqlite_exec(). For a normal halt, this should be SQLITE_OK (0). For errors, it can be some other value. The operand P2 is only used when there is an error. There is an implied "Halt 0 0 0" instruction at the end of every program, which the VDBE appends when it prepares a program to run.

Tracing VDBE Program Execution

If the SQLite library is compiled without the NDEBUG preprocessor macro, then the PRAGMA vdbe_trace causes the VDBE to trace the execution of programs. Though this feature was originally intended for testing and debugging, it can also be useful in learning about how the VDBE operates. Use "`PRAGMA vdbe_trace=ON;`" to turn tracing on and "`PRAGMA vdbe_trace=OFF`" to turn tracing back off. Like this:

```
sqlite> PRAGMA vdbe_trace=ON;
   0 Halt            0    0
sqlite> INSERT INTO examp VALUES('Hello, World!',99);
   0 Transaction     0    0
   1 VerifyCookie    0   81
   2 Transaction     1    0
   3 Integer         0    0
Stack: i:0
   4 OpenWrite       0    3 examp
   5 NewRecno        0    0
Stack: i:2
   6 String          0    0 Hello, World!
```

```
Stack: t[Hello,.World!] i:2
   7 Integer        99    0 99
Stack: si:99 t[Hello,.World!] i:2
   8 MakeRecord      2    0
Stack: s[...Hello,.World!.99] i:2
   9 PutIntKey       0    1
  10 Close           0    0
  11 Commit          0    0
  12 Halt            0    0
```

With tracing mode on, the VDBE prints each instruction prior to executing it. After the instruction is executed, the top few entries in the stack are displayed. The stack display is omitted if the stack is empty.

On the stack display, most entries are shown with a prefix that tells the datatype of that stack entry. Integers begin with "`i:`". Floating point values begin with "`r:`". (The "r" stands for "real-number".) Strings begin with either "`s:`", "`t:`", "`e:`" or "`z:`". The difference among the string prefixes is caused by how their memory is allocated. The z: strings are stored in memory obtained from **malloc()**. The t: strings are statically allocated. The e: strings are ephemeral. All other strings have the s: prefix. This doesn't make any difference to you, the observer, but it is vitally important to the VDBE since the z: strings need to be passed to **free()** when they are popped to avoid a memory leak. Note that only the first 10 characters of string values are displayed and that binary values (such as the result of the MakeRecord instruction) are treated as strings. The only other datatype that can be stored on the VDBE stack is a NULL, which is display without prefix as simply "`NULL`". If an integer has been placed on the stack as both an integer and a string, its prefix is "`si:`".

Simple Queries

At this point, you should understand the basics of how the VDBE writes to a database. Now let's look at how it does queries. We will use the following simple SELECT statement as our example:

```
SELECT * FROM examp;
```

The VDBE program generated for this SQL statement is as follows:

```
sqlite> EXPLAIN SELECT * FROM examp;
addr  opcode        p1     p2     p3
----  ------------  -----  -----  ----------------------------------
0     ColumnName    0      0      one
1     ColumnName    1      0      two
2     Integer       0      0
3     OpenRead      0      3      examp
4     VerifyCookie  0      81
5     Rewind        0      10
```

```
6     Column        0     0
7     Column        0     1
8     Callback      2     0
9     Next          0     6
10    Close         0     0
11    Halt          0     0
```

Before we begin looking at this problem, let's briefly review how queries work in SQLite so that we will know what we are trying to accomplish. For each row in the result of a query, SQLite will invoke a callback function with the following prototype:

```
int Callback(void *pUserData, int nColumn, char *azData[], char *azColumnName[]);
```

The SQLite library supplies the VDBE with a pointer to the callback function and the **pUserData** pointer. (Both the callback and the user data were originally passed in as arguments to the **sqlite_exec()** API function.) The job of the VDBE is to come up with values for **nColumn**, **azData[]**, and **azColumnName[]**. **nColumn** is the number of columns in the results, of course. **azColumnName[]** is an array of strings where each string is the name of one of the result columns. **azData[]** is an array of strings holding the actual data.

```
0     ColumnName    0     0     one
1     ColumnName    1     0     two
```

The first two instructions in the VDBE program for our query are concerned with setting up values for **azColumn**. The ColumnName instructions tell the VDBE what values to fill in for each element of the **azColumnName[]** array. Every query will begin with one ColumnName instruction for each column in the result, and there will be a matching Column instruction for each one later in the query.

```
2     Integer       0     0
3     OpenRead      0     3     examp
4     VerifyCookie  0     81
```

Instructions 2 and 3 open a read cursor on the database table that is to be queried. This works the same as the OpenWrite instruction in the INSERT example except that the cursor is opened for reading this time instead of for writing. Instruction 4 verifies the database schema as in the INSERT example.

```
5     Rewind        0     10
```

The Rewind instruction initializes a loop that iterates over the "examp" table. It rewinds the cursor P1 to the first entry in its table. This is required by the Column and Next instructions, which use the cursor to iterate through the table. If the table is empty, then jump to P2 (10), which is the instruction just past the loop. If the table is not empty, fall through to the following instruction at 6, which is the beginning of the loop body.

```
6     Column        0      0
7     Column        0      1
8     Callback      2      0
```

The instructions 6 through 8 form the body of the loop that will execute once for each record in the database file. The Column instructions at addresses 6 and 7 each take the P2-th column from the P1-th cursor and push it onto the stack. In this example, the first Column instruction is pushing the value for the column "one" onto the stack and the second Column instruction is pushing the value for column "two". The Callback instruction at address 8 invokes the callback() function. The P1 operand to Callback becomes the value for **nColumn**. The Callback instruction pops P1 values from the stack and uses them to fill the **azData[]** array.

```
9     Next          0      6
```

The instruction at address 9 implements the branching part of the loop. Together with the Rewind at address 5 it forms the loop logic. This is a key concept that you should pay close attention to. The Next instruction advances the cursor P1 to the next record. If the cursor advance was successful, then jump immediately to P2 (6, the beginning of the loop body). If the cursor was at the end, then fall through to the following instruction, which ends the loop.

```
10    Close         0      0
11    Halt          0      0
```

The Close instruction at the end of the program closes the cursor that points into the table "examp". It is not really necessary to call Close here since all cursors will be automatically closed by the VDBE when the program halts. But we needed an instruction for the Rewind to jump to so we might as well go ahead and have that instruction do something useful. The Halt instruction ends the VDBE program.

Note that the program for this SELECT query didn't contain the Transaction and Commit instructions used in the INSERT example. Because the SELECT is a read operation that doesn't alter the database, it doesn't require a transaction.

A Slightly More Complex Query

The key points of the previous example were the use of the Callback instruction to invoke the callback function, and the use of the Next instruction to implement a loop over all records of the database file. This example attempts to drive home those ideas by demonstrating a slightly more complex query that involves more columns of output, some of which are computed values, and a WHERE clause that limits which records actually make it to the callback function. Consider this query:

```
SELECT one, two, one || two AS 'both'
FROM examp
WHERE one LIKE 'H%'
```

This query is perhaps a bit contrived, but it does serve to illustrate our points. The result will have three column with names "one", "two", and "both". The first two columns are direct copies of the two columns in the table and the third result column is a string formed by concatenating the first and second columns of the table. Finally, the WHERE clause says that we will only chose rows for the results where the "one" column begins with an "H". Here is what the VDBE program looks like for this query:

```
addr  opcode        p1     p2     p3
----  ------------  -----  -----  ----------------------------------
0     ColumnName    0      0      one
1     ColumnName    1      0      two
2     ColumnName    2      0      both
3     Integer       0      0
4     OpenRead      0      3      examp
5     VerifyCookie  0      81
6     Rewind        0      18
7     String        0      0      H%
8     Column        0      0
9     Function      2      0      ptr(0x7f1ac0)
10    IfNot         1      17
11    Column        0      0
12    Column        0      1
13    Column        0      0
14    Column        0      1
15    Concat        2      0
16    Callback      3      0
17    Next          0      7
18    Close         0      0
19    Halt          0      0
```

Except for the WHERE clause, the structure of the program for this example is very much like the prior example, just with an extra column. There are now 3 columns, instead of 2 as before, and there are three ColumnName instructions. A cursor is opened using the OpenRead instruction, just like in the prior example. The Rewind instruction at address 6 and the Next at address 17 form a loop over all records of the table. The Close instruction at the end is there to give the Rewind instruction something to jump to when it is done. All of this is just like in the first query demonstration.

The Callback instruction in this example has to generate data for three result columns instead of two, but is otherwise the same as in the first query. When the Callback instruction is invoked, the left-most column of the result should be the lowest in the stack and the right-most result column should be the top of the stack. We can see the stack being set up this way at addresses 11 through 15. The Column instructions at 11 and 12 push the values for the first two columns in the result. The two Column instructions at 13 and 14 pull in the values needed to compute the third

result column and the Concat instruction at 15 joins them together into a single entry on the stack.

The only thing that is really new about the current example is the WHERE clause which is implemented by instructions at addresses 7 through 10. Instructions at address 7 and 8 push onto the stack the value of the "one" column from the table and the literal string "H%". The Function instruction at address 9 pops these two values from the stack and pushes the result of the LIKE() function back onto the stack. The IfNot instruction pops the top stack value and causes an immediate jump forward to the Next instruction if the top value was false (*not* not like the literal string "H%"). Taking this jump effectively skips the callback, which is the whole point of the WHERE clause. If the result of the comparison is true, the jump is not taken and control falls through to the Callback instruction below.

Notice how the LIKE operator is implemented. It is a user-defined function in SQLite, so the address of its function definition is specified in P3. The operand P1 is the number of function arguments for it to take from the stack. In this case the LIKE() function takes 2 arguments. The arguments are taken off the stack in reverse order (right-to-left), so the pattern to match is the top stack element, and the next element is the data to compare. The return value is pushed onto the stack.

A Template For SELECT Programs

The first two query examples illustrate a kind of template that every SELECT program will follow. Basically, we have:

1. Initialize the **azColumnName[]** array for the callback.
2. Open a cursor into the table to be queried.
3. For each record in the table, do:
    a. If the WHERE clause evaluates to FALSE, then skip the steps that follow and continue to the next record.
    b. Compute all columns for the current row of the result.
    c. Invoke the callback function for the current row of the result.
4. Close the cursor.

This template will be expanded considerably as we consider additional complications such as joins, compound selects, using indices to speed the search, sorting, and aggregate functions with and without GROUP BY and HAVING clauses. But the same basic ideas will continue to apply.

UPDATE And DELETE Statements

The UPDATE and DELETE statements are coded using a template that is very similar to the SELECT statement template. The main difference, of course, is that the end action is to modify the database rather than invoke a callback function. Because it modifies the database it will also use transactions. Let's begin by looking at a DELETE statement:

```
DELETE FROM examp WHERE two<50;
```

This DELETE statement will remove every record from the "examp" table where the "two" column is less than 50. The code generated to do this is as follows:

```
addr  opcode        p1     p2     p3
----  ------------  -----  -----  ----------------------------------
0     Transaction   1      0
1     Transaction   0      0
2     VerifyCookie  0      178
3     Integer       0      0
4     OpenRead      0      3      examp
5     Rewind        0      12
6     Column        0      1
7     Integer       50     0      50
8     Ge            1      11
9     Recno         0      0
10    ListWrite     0      0
11    Next          0      6
12    Close         0      0
13    ListRewind    0      0
14    Integer       0      0
15    OpenWrite     0      3
16    ListRead      0      20
17    NotExists     0      19
18    Delete        0      1
19    Goto          0      16
20    ListReset     0      0
21    Close         0      0
22    Commit        0      0
23    Halt          0      0
```

Here is what the program must do. First it has to locate all of the records in the table "examp" that are to be deleted. This is done using a loop very much like the loop used in the SELECT examples above. Once all records have been located, then we can go back through and delete them one by one. Note that we cannot delete each record as soon as we find it. We have to locate all records first, then go back and delete them. This is because the SQLite database backend might change the scan order after a delete operation. And if the scan order changes in the middle of the scan, some records might be visited more than once and other records might not be visited at all.

So the implementation of DELETE is really in two loops. The first loop (instructions 5 through 11) locates the records that are to be deleted and saves their keys onto a temporary list, and the second loop (instructions 16 through 19) uses the key list to delete the records one by one.

```
0     Transaction   1      0
1     Transaction   0      0
2     VerifyCookie  0      178
3     Integer       0      0
4     OpenRead      0      3      examp
```

Instructions 0 though 4 are as in the INSERT example. They start transactions for the main and temporary databases, verify the database schema for the main database, and open a read cursor on the table "examp". Notice that the cursor is opened for reading, not writing. At this stage of the program we are only going to be scanning the table, not changing it. We will reopen the same table for writing later, at instruction 15.

```
5     Rewind        0      12
```

As in the SELECT example, the Rewind instruction rewinds the cursor to the beginning of the table, readying it for use in the loop body.

```
6     Column        0      1
7     Integer       50     0      50
8     Ge            1      11
```

The WHERE clause is implemented by instructions 6 through 8. The job of the where clause is to skip the ListWrite if the WHERE condition is false. To this end, it jumps ahead to the Next instruction if the "two" column (extracted by the Column instruction) is greater than or equal to 50.

As before, the Column instruction uses cursor P1 and pushes the data record in column P2 (1, column "two") onto the stack. The Integer instruction pushes the value 50 onto the top of the stack. After these two instructions the stack looks like:

(integer) 50
(record) current record for column "two"

The Ge operator compares the top two elements on the stack, pops them, and then branches based on the result of the comparison. If the second element is >= the top element, then jump to address P2 (the Next instruction at the end of the loop). Because P1 is true, if either operand is NULL (and thus the result is NULL) then take the jump. If we don't jump, just advance to the next instruction.

```
9     Recno         0      0
10    ListWrite     0      0
```

The Recno instruction pushes onto the stack an integer which is the first 4 bytes of the key to the current entry in a sequential scan of the table pointed to by cursor P1. The ListWrite instruction writes the integer on the top of the stack into a temporary storage list and pops the top element. This is the important work of this

loop, to store the keys of the records to be deleted so we can delete them in the second loop. After this ListWrite instruction the stack is empty again.

```
11    Next          0      6
12    Close         0      0
```

The Next instruction increments the cursor to point to the next element in the table pointed to by cursor P0, and if it was successful branches to P2 (6, the beginning of the loop body). The Close instruction closes cursor P1. It doesn't affect the temporary storage list because it isn't associated with cursor P1; it is instead a global working list (which can be saved with ListPush).

```
13    ListRewind    0      0
```

The ListRewind instruction rewinds the temporary storage list to the beginning. This prepares it for use in the second loop.

```
14    Integer       0      0
15    OpenWrite     0      3
```

As in the INSERT example, we push the database number P1 (0, the main database) onto the stack and use OpenWrite to open the cursor P1 on table P2 (base page 3, "examp") for modification.

```
16    ListRead      0      20
17    NotExists     0      19
18    Delete        0      1
19    Goto          0      16
```

This loop does the actual deleting. It is organized differently from the one in the UPDATE example. The ListRead instruction plays the role that the Next did in the INSERT loop, but because it jumps to P2 on failure, and Next jumps on success, we put it at the start of the loop instead of the end. This means that we have to put a Goto at the end of the loop to jump back to the loop test at the beginning. So this loop has the form of a C while(){...} loop, while the loop in the INSERT example had the form of a do{...}while() loop. The Delete instruction fills the role that the callback function did in the preceding examples.

The ListRead instruction reads an element from the temporary storage list and pushes it onto the stack. If this was successful, it continues to the next instruction. If this fails because the list is empty, it branches to P2, which is the instruction just after the loop. Afterwards the stack looks like:

(integer) key for current record

Notice the similarity between the ListRead and Next instructions. Both operations work according to this rule:

> Push the next "thing" onto the stack and fall through OR jump to P2, depending on whether or not there is a next "thing" to push.

One difference between Next and ListRead is their idea of a "thing". The "things" for the Next instruction are records in a database file. "Things" for ListRead are integer keys in a list. Another difference is whether to jump or fall through if there is no next "thing". In this case, Next falls through, and ListRead jumps. Later on, we will see other looping instructions (NextIdx and SortNext) that operate using the same principle.

The NotExists instruction pops the top stack element and uses it as an integer key. If a record with that key does not exist in table P1, then jump to P2. If a record does exist, then fall through to the next instruction. In this case P2 takes us to the Goto at the end of the loop, which jumps back to the ListRead at the beginning. This could have been coded to have P2 be 16, the ListRead at the start of the loop, but the SQLite parser which generated this code didn't make that optimization.

The Delete does the work of this loop; it pops an integer key off the stack (placed there by the preceding ListRead) and deletes the record of cursor P1 that has that key. Because P2 is true, the row change counter is incremented.

The Goto jumps back to the beginning of the loop. This is the end of the loop.

```
20    ListReset     0      0
21    Close         0      0
22    Commit        0      0
23    Halt          0      0
```

This block of instruction cleans up the VDBE program. Three of these instructions aren't really required, but are generated by the SQLite parser from its code templates, which are designed to handle more complicated cases.

The ListReset instruction empties the temporary storage list. This list is emptied automatically when the VDBE program terminates, so it isn't necessary in this case. The Close instruction closes the cursor P1. Again, this is done by the VDBE engine when it is finished running this program. The Commit ends the current transaction successfully, and causes all changes that occurred in this transaction to be saved to the database. The final Halt is also unnecessary, since it is added to every VDBE program when it is prepared to run.

UPDATE statements work very much like DELETE statements except that instead of deleting the record they replace it with a new one. Consider this example:

```
UPDATE examp SET one= '(' || one || ')' WHERE two < 50;
```

Instead of deleting records where the "two" column is less than 50, this statement just puts the "one" column in parentheses The VDBE program to implement this statement follows:

```
addr  opcode        p1     p2     p3
----  ------------  -----  -----  ----------------------------------
0     Transaction   1      0
1     Transaction   0      0
2     VerifyCookie  0      178
3     Integer       0      0
4     OpenRead      0      3      examp
5     Rewind        0      12
6     Column        0      1
7     Integer       50     0      50
8     Ge            1      11
9     Recno         0      0
10    ListWrite     0      0
11    Next          0      6
12    Close         0      0
13    Integer       0      0
14    OpenWrite     0      3
15    ListRewind    0      0
16    ListRead      0      28
17    Dup           0      0
18    NotExists     0      16
19    String        0      0      (
20    Column        0      0
21    Concat        2      0
22    String        0      0      )
23    Concat        2      0
24    Column        0      1
25    MakeRecord    2      0
26    PutIntKey     0      1
27    Goto          0      16
28    ListReset     0      0
29    Close         0      0
30    Commit        0      0
31    Halt          0      0
```

This program is essentially the same as the DELETE program except that the body of the second loop has been replace by a sequence of instructions (at addresses 17 through 26) that update the record rather than delete it. Most of this instruction sequence should already be familiar to you, but there are a couple of minor twists so we will go over it briefly. Also note that the order of some of the instructions before and after the 2nd loop has changed. This is just the way the SQLite parser chose to output the code using a different template.

As we enter the interior of the second loop (at instruction 17) the stack contains a single integer which is the key of the record we want to modify. We are going to need to use this key twice: once to fetch the old value of the record and a second time to write back the revised record. So the first instruction is a Dup to make a

duplicate of the key on the top of the stack. The Dup instruction will duplicate any element of the stack, not just the top element. You specify which element to duplication using the P1 operand. When P1 is 0, the top of the stack is duplicated. When P1 is 1, the next element down on the stack duplication. And so forth.

After duplicating the key, the next instruction, NotExists, pops the stack once and uses the value popped as a key to check the existence of a record in the database file. If there is no record for this key, it jumps back to the ListRead to get another key.

Instructions 19 through 25 construct a new database record that will be used to replace the existing record. This is the same kind of code that we saw in the description of INSERT and will not be described further. After instruction 25 executes, the stack looks like this:

(record) new data record
(integer) key

The PutIntKey instruction (also described during the discussion about INSERT) writes an entry into the database file whose data is the top of the stack and whose key is the next on the stack, and then pops the stack twice. The PutIntKey instruction will overwrite the data of an existing record with the same key, which is what we want here. Overwriting was not an issue with INSERT because with INSERT the key was generated by the NewRecno instruction which is guaranteed to provide a key that has not been used before.

CREATE and DROP

Using CREATE or DROP to create or destroy a table or index is really the same as doing an INSERT or DELETE from the special "sqlite_master" table, at least from the point of view of the VDBE. The sqlite_master table is a special table that is automatically created for every SQLite database. It looks like this:

```
CREATE TABLE sqlite_master (
  type      TEXT,    -- either "table" or "index"
  name      TEXT,    -- name of this table or index
  tbl_name  TEXT,    -- for indices: name of associated table
  sql       TEXT     -- SQL text of the original CREATE statement
)
```

Every table (except the "sqlite_master" table itself) and every named index in an SQLite database has an entry in the sqlite_master table. You can query this table using a SELECT statement just like any other table. But you are not allowed to directly change the table using UPDATE, INSERT, or DELETE. Changes to sqlite_master have to occur using the CREATE and DROP commands because SQLite also has to update some of its internal data structures when tables and indices are added or destroyed.

But from the point of view of the VDBE, a CREATE works pretty much like an INSERT and a DROP works like a DELETE. When the SQLite library opens to an existing database, the first thing it does is a SELECT to read the "sql" columns from all entries of the sqlite_master table. The "sql" column contains the complete SQL text of the CREATE statement that originally generated the index or table. This text is fed back into the SQLite parser and used to reconstruct the internal data structures describing the index or table.

Using Indexes To Speed Searching

In the example queries above, every row of the table being queried must be loaded off of the disk and examined, even if only a small percentage of the rows end up in the result. This can take a long time on a big table. To speed things up, SQLite can use an index.

An SQLite file associates a key with some data. For an SQLite table, the database file is set up so that the key is an integer and the data is the information for one row of the table. Indices in SQLite reverse this arrangement. The index key is (some of) the information being stored and the index data is an integer. To access a table row that has some particular content, we first look up the content in the index table to find its integer index, then we use that integer to look up the complete record in the table.

Note that SQLite uses b-trees, which are a sorted data structure, so indices can be used when the WHERE clause of the SELECT statement contains tests for equality or inequality. Queries like the following can use an index if it is available:

```
SELECT * FROM examp WHERE two=50;
SELECT * FROM examp WHERE two<50;
SELECT * FROM examp WHERE two IN (50, 100);
```

If there exists an index that maps the "two" column of the "examp" table into integers, then SQLite will use that index to find the integer keys of all rows in examp that have a value of 50 for column two, or all rows that are less than 50, etc. But the following queries cannot use the index:

```
SELECT * FROM examp WHERE two%50 = 10;
SELECT * FROM examp WHERE two&127 = 3;
```

Note that the SQLite parser will not always generate code to use an index, even if it is possible to do so. The following queries will not currently use the index:

```
SELECT * FROM examp WHERE two+10 = 50;
SELECT * FROM examp WHERE two=50 OR two=100;
```

To understand better how indices work, lets first look at how they are created. Let's go ahead and put an index on the two column of the examp table. We have:

```
CREATE INDEX examp_idx1 ON examp(two);
```

The VDBE code generated by the above statement looks like the following:

```
addr  opcode        p1     p2     p3
----  ------------  -----  -----  ----------------------------------
0     Transaction   1      0
1     Transaction   0      0
2     VerifyCookie  0      178
3     Integer       0      0
4     OpenWrite     0      2
5     NewRecno      0      0
6     String        0      0      index
7     String        0      0      examp_idx1
8     String        0      0      examp
9     CreateIndex   0      0      ptr(0x791380)
10    Dup           0      0
11    Integer       0      0
12    OpenWrite     1      0
13    String        0      0      CREATE INDEX examp_idx1 ON examp(tw
14    MakeRecord    5      0
15    PutIntKey     0      0
16    Integer       0      0
17    OpenRead      2      3      examp
18    Rewind        2      24
19    Recno         2      0
20    Column        2      1
21    MakeIdxKey    1      0      n
22    IdxPut        1      0      indexed columns are not unique
23    Next          2      19
24    Close         2      0
25    Close         1      0
26    Integer       333    0
27    SetCookie     0      0
28    Close         0      0
29    Commit        0      0
30    Halt          0      0
```

Remember that every table (except sqlite_master) and every named index has an entry in the sqlite_master table. Since we are creating a new index, we have to add a new entry to sqlite_master. This is handled by instructions 3 through 15. Adding an entry to sqlite_master works just like any other INSERT statement so we will not say any more about it here. In this example, we want to focus on populating the new index with valid data, which happens on instructions 16 through 23.

```
16    Integer       0      0
17    OpenRead      2      3      examp
```

The first thing that happens is that we open the table being indexed for reading. In order to construct an index for a table, we have to know what is in that table. The index has already been opened for writing using cursor 0 by instructions 3 and 4.

```
18    Rewind        2     24
19    Recno         2     0
20    Column        2     1
21    MakeIdxKey    1     0     n
22    IdxPut        1     0     indexed columns are not unique
23    Next          2     19
```

Instructions 18 through 23 implement a loop over every row of the table being indexed. For each table row, we first extract the integer key for that row using Recno in instruction 19, then get the value of the "two" column using Column in instruction 20. The MakeIdxKey instruction at 21 converts data from the "two" column (which is on the top of the stack) into a valid index key. For an index on a single column, this is basically a no-op. But if the P1 operand to MakeIdxKey had been greater than one multiple entries would have been popped from the stack and converted into a single index key. The IdxPut instruction at 22 is what actually creates the index entry. IdxPut pops two elements from the stack. The top of the stack is used as a key to fetch an entry from the index table. Then the integer which was second on stack is added to the set of integers for that index and the new record is written back to the database file. Note that the same index entry can store multiple integers if there are two or more table entries with the same value for the two column.

Now let's look at how this index will be used. Consider the following query:

```
SELECT * FROM examp WHERE two==50;
```

SQLite generates the following VDBE code to handle this query:

```
addr  opcode        p1    p2    p3
----  ------------  -----  -----  ----------------------------------
0     ColumnName    0     0     one
1     ColumnName    1     0     two
2     Integer       0     0
3     OpenRead      0     3     examp
4     VerifyCookie  0     256
5     Integer       0     0
6     OpenRead      1     4     examp_idx1
7     Integer       50    0     50
8     MakeKey       1     0     n
9     MemStore      0     0
10    MoveTo        1     19
11    MemLoad       0     0
12    IdxGT         1     19
13    IdxRecno      1     0
```

```
14    MoveTo        0      0
15    Column        0      0
16    Column        0      1
17    Callback      2      0
18    Next          1      11
19    Close         0      0
20    Close         1      0
21    Halt          0      0
```

The SELECT begins in a familiar fashion. First the column names are initialized and the table being queried is opened. Things become different beginning with instructions 5 and 6 where the index file is also opened. Instructions 7 and 8 make a key with the value of 50. The MemStore instruction at 9 stores the index key in VDBE memory location 0. The VDBE memory is used to avoid having to fetch a value from deep in the stack, which can be done, but makes the program harder to generate. The following instruction MoveTo at address 10 pops the key off the stack and moves the index cursor to the first row of the index with that key. This initializes the cursor for use in the following loop.

Instructions 11 through 18 implement a loop over all index records with the key that was fetched by instruction 8. All of the index records with this key will be contiguous in the index table, so we walk through them and fetch the corresponding table key from the index. This table key is then used to move the cursor to that row in the table. The rest of the loop is the same as the loop for the non-indexed SELECT query.

The loop begins with the MemLoad instruction at 11 which pushes a copy of the index key back onto the stack. The instruction IdxGT at 12 compares the key to the key in the current index record pointed to by cursor P1. If the index key at the current cursor location is greater than the index we are looking for, then jump out of the loop.

The instruction IdxRecno at 13 pushes onto the stack the table record number from the index. The following MoveTo pops it and moves the table cursor to that row. The next 3 instructions select the column data the same way as in the non- indexed case. The Column instructions fetch the column data and the callback function is invoked. The final Next instruction advances the index cursor, not the table cursor, to the next row, and then branches back to the start of the loop if there are any index records left.

Since the index is used to look up values in the table, it is important that the index and table be kept consistent. Now that there is an index on the examp table, we will have to update that index whenever data is inserted, deleted, or changed in the examp table. Remember the first example above where we were able to insert a new row into the "examp" table using 12 VDBE instructions. Now that this table is indexed, 19 instructions are required. The SQL statement is this:

```
INSERT INTO examp VALUES('Hello, World!',99);
```

And the generated code looks like this:

```
addr  opcode        p1     p2     p3
----  ------------  -----  -----  ----------------------------------
0     Transaction   1      0
1     Transaction   0      0
2     VerifyCookie  0      256
3     Integer       0      0
4     OpenWrite     0      3      examp
5     Integer       0      0
6     OpenWrite     1      4      examp_idx1
7     NewRecno      0      0
8     String        0      0      Hello, World!
9     Integer       99     0      99
10    Dup           2      1
11    Dup           1      1
12    MakeIdxKey    1      0      n
13    IdxPut        1      0
14    MakeRecord    2      0
15    PutIntKey     0      1
16    Close         0      0
17    Close         1      0
18    Commit        0      0
19    Halt          0      0
```

At this point, you should understand the VDBE well enough to figure out on your own how the above program works. So we will not discuss it further in this text.

Joins

In a join, two or more tables are combined to generate a single result. The result table consists of every possible combination of rows from the tables being joined. The easiest and most natural way to implement this is with nested loops.

Recall the query template discussed above where there was a single loop that searched through every record of the table. In a join we have basically the same thing except that there are nested loops. For example, to join two tables, the query template might look something like this:

1. Initialize the **azColumnName[]** array for the callback.
2. Open two cursors, one to each of the two tables being queried.
3. For each record in the first table, do:
    a. For each record in the second table do:
        i. If the WHERE clause evaluates to FALSE, then skip the steps that follow and continue to the next record.
        ii. Compute all columns for the current row of the result.
        iii. Invoke the callback function for the current row of the result.

4. Close both cursors.

This template will work, but it is likely to be slow since we are now dealing with an $O(N^2)$ loop. But it often works out that the WHERE clause can be factored into terms and that one or more of those terms will involve only columns in the first table. When this happens, we can factor part of the WHERE clause test out of the inner loop and gain a lot of efficiency. So a better template would be something like this:

1. Initialize the **azColumnName[]** array for the callback.
2. Open two cursors, one to each of the two tables being queried.
3. For each record in the first table, do:
    a. Evaluate terms of the WHERE clause that only involve columns from the first table. If any term is false (meaning that the whole WHERE clause must be false) then skip the rest of this loop and continue to the next record.
    b. For each record in the second table do:
        i. If the WHERE clause evaluates to FALSE, then skip the steps that follow and continue to the next record.
        ii. Compute all columns for the current row of the result.
        iii. Invoke the callback function for the current row of the result.
4. Close both cursors.

Additional speed-up can occur if an index can be used to speed the search of either or the two loops.

SQLite always constructs the loops in the same order as the tables appear in the FROM clause of the SELECT statement. The left-most table becomes the outer loop and the right-most table becomes the inner loop. It is possible, in theory, to reorder the loops in some circumstances to speed the evaluation of the join. But SQLite does not attempt this optimization.

You can see how SQLite constructs nested loops in the following example:

```
CREATE TABLE examp2(three int, four int);
SELECT * FROM examp, examp2 WHERE two<50 AND four==two;

addr  opcode        p1     p2     p3
----  ------------  -----  -----  ----------------------------------
0     ColumnName    0      0      examp.one
1     ColumnName    1      0      examp.two
2     ColumnName    2      0      examp2.three
3     ColumnName    3      0      examp2.four
4     Integer       0      0
5     OpenRead      0      3      examp
6     VerifyCookie  0      909
7     Integer       0      0
8     OpenRead      1      5      examp2
9     Rewind        0      24
10    Column        0      1
```

```
11    Integer       50     0      50
12    Ge            1      23
13    Rewind        1      23
14    Column        1      1
15    Column        0      1
16    Ne            1      22
17    Column        0      0
18    Column        0      1
19    Column        1      0
20    Column        1      1
21    Callback      4      0
22    Next          1      14
23    Next          0      10
24    Close         0      0
25    Close         1      0
26    Halt          0      0
```

The outer loop over table examp is implement by instructions 7 through 23. The inner loop is instructions 13 through 22. Notice that the "two<50" term of the WHERE expression involves only columns from the first table and can be factored out of the inner loop. SQLite does this and implements the "two<50" test in instructions 10 through 12. The "four==two" test is implement by instructions 14 through 16 in the inner loop.

SQLite does not impose any arbitrary limits on the tables in a join. It also allows a table to be joined with itself.

The ORDER BY clause

For historical reasons, and for efficiency, all sorting is currently done in memory.

SQLite implements the ORDER BY clause using a special set of instructions to control an object called a sorter. In the inner-most loop of the query, where there would normally be a Callback instruction, instead a record is constructed that contains both callback parameters and a key. This record is added to the sorter (in a linked list). After the query loop finishes, the list of records is sorted and this list is walked. For each record on the list, the callback is invoked. Finally, the sorter is closed and memory is deallocated.

We can see the process in action in the following query:

```
SELECT * FROM examp ORDER BY one DESC, two;

addr  opcode        p1     p2     p3
----  ------------  -----  -----  ----------------------------------
0     ColumnName    0      0      one
1     ColumnName    1      0      two
```

```
2     Integer       0      0
3     OpenRead      0      3      examp
4     VerifyCookie  0      909
5     Rewind        0      14
6     Column        0      0
7     Column        0      1
8     SortMakeRec   2      0
9     Column        0      0
10    Column        0      1
11    SortMakeKey   2      0      D+
12    SortPut       0      0
13    Next          0      6
14    Close         0      0
15    Sort          0      0
16    SortNext      0      19
17    SortCallback  2      0
18    Goto          0      16
19    SortReset     0      0
20    Halt          0      0
```

There is only one sorter object, so there are no instructions to open or close it. It is opened automatically when needed, and it is closed when the VDBE program halts.

The query loop is built from instructions 5 through 13. Instructions 6 through 8 build a record that contains the azData[] values for a single invocation of the callback. A sort key is generated by instructions 9 through 11. Instruction 12 combines the invocation record and the sort key into a single entry and puts that entry on the sort list.

The P3 argument of instruction 11 is of particular interest. The sort key is formed by prepending one character from P3 to each string and concatenating all the strings. The sort comparison function will look at this character to determine whether the sort order is ascending or descending, and whether to sort as a string or number. In this example, the first column should be sorted as a string in descending order so its prefix is "D" and the second column should sorted numerically in ascending order so its prefix is "+". Ascending string sorting uses "A", and descending numeric sorting uses "-".

After the query loop ends, the table being queried is closed at instruction 14. This is done early in order to allow other processes or threads to access that table, if desired. The list of records that was built up inside the query loop is sorted by the instruction at 15. Instructions 16 through 18 walk through the record list (which is now in sorted order) and invoke the callback once for each record. Finally, the sorter is closed at instruction 19.

Aggregate Functions And The GROUP BY and HAVING Clauses

To compute aggregate functions, the VDBE implements a special data structure and instructions for controlling that data structure. The data structure is an unordered set of buckets, where each bucket has a key and one or more memory locations. Within the query loop, the GROUP BY clause is used to construct a key and the bucket with that key is brought into focus. A new bucket is created with the key if one did not previously exist. Once the bucket is in focus, the memory locations of the bucket are used to accumulate the values of the various aggregate functions. After the query loop terminates, each bucket is visited once to generate a single row of the results.

An example will help to clarify this concept. Consider the following query:

```
SELECT three, min(three+four)+avg(four)
FROM examp2
GROUP BY three;
```

The VDBE code generated for this query is as follows:

```
addr  opcode        p1     p2     p3
----  ------------  -----  -----  ----------------------------------
0     ColumnName    0      0      three
1     ColumnName    1      0      min(three+four)+avg(four)
2     AggReset      0      3
3     AggInit       0      1      ptr(0x7903a0)
4     AggInit       0      2      ptr(0x790700)
5     Integer       0      0
6     OpenRead      0      5      examp2
7     VerifyCookie  0      909
8     Rewind        0      23
9     Column        0      0
10    MakeKey       1      0      n
11    AggFocus      0      14
12    Column        0      0
13    AggSet        0      0
14    Column        0      0
15    Column        0      1
16    Add           0      0
17    Integer       1      0
18    AggFunc       0      1      ptr(0x7903a0)
19    Column        0      1
20    Integer       2      0
21    AggFunc       0      1      ptr(0x790700)
22    Next          0      9
23    Close         0      0
24    AggNext       0      31
25    AggGet        0      0
26    AggGet        0      1
```

```
27    AggGet        0      2
28    Add           0      0
29    Callback      2      0
30    Goto          0      24
31    Noop          0      0
32    Halt          0      0
```

The first instruction of interest is the AggReset at 2. The AggReset instruction initializes the set of buckets to be the empty set and specifies the number of memory slots available in each bucket as P2. In this example, each bucket will hold 3 memory slots. It is not obvious, but if you look closely at the rest of the program you can figure out what each of these slots is intended for.

Memory Slot	Intended Use Of This Memory Slot
0	The "three" column -- the key to the bucket
1	The minimum "three+four" value
2	The sum of all "four" values. This is used to compute "avg(four)".

The query loop is implemented by instructions 8 through 22. The aggregate key specified by the GROUP BY clause is computed by instructions 9 and 10. Instruction 11 causes the appropriate bucket to come into focus. If a bucket with the given key does not already exists, a new bucket is created and control falls through to instructions 12 and 13 which initialize the bucket. If the bucket does already exist, then a jump is made to instruction 14. The values of aggregate functions are updated by the instructions between 11 and 21. Instructions 14 through 18 update memory slot 1 to hold the next value "min(three+four)". Then the sum of the "four" column is updated by instructions 19 through 21.

After the query loop is finished, the table "examp2" is closed at instruction 23 so that its lock will be released and it can be used by other threads or processes. The next step is to loop over all aggregate buckets and output one row of the result for each bucket. This is done by the loop at instructions 24 through 30. The AggNext instruction at 24 brings the next bucket into focus, or jumps to the end of the loop if all buckets have been examined already. The 3 columns of the result are fetched from the aggregator bucket in order at instructions 25 through 27. Finally, the callback is invoked at instruction 29.

In summary then, any query with aggregate functions is implemented by two loops. The first loop scans the input table and computes aggregate information into buckets and the second loop scans through all the buckets to compute the final result.

The realization that an aggregate query is really two consecutive loops makes it much easier to understand the difference between a WHERE clause and a HAVING

clause in SQL query statement. The WHERE clause is a restriction on the first loop and the HAVING clause is a restriction on the second loop. You can see this by adding both a WHERE and a HAVING clause to our example query:

```
SELECT three, min(three+four)+avg(four)
FROM examp2
WHERE three>four
GROUP BY three
HAVING avg(four)<10;

addr  opcode        p1     p2     p3
----  ------------  -----  -----  ----------------------------------
0     ColumnName    0      0      three
1     ColumnName    1      0      min(three+four)+avg(four)
2     AggReset      0      3
3     AggInit       0      1      ptr(0x7903a0)
4     AggInit       0      2      ptr(0x790700)
5     Integer       0      0
6     OpenRead      0      5      examp2
7     VerifyCookie  0      909
8     Rewind        0      26
9     Column        0      0
10    Column        0      1
11    Le            1      25
12    Column        0      0
13    MakeKey       1      0      n
14    AggFocus      0      17
15    Column        0      0
16    AggSet        0      0
17    Column        0      0
18    Column        0      1
19    Add           0      0
20    Integer       1      0
21    AggFunc       0      1      ptr(0x7903a0)
22    Column        0      1
23    Integer       2      0
24    AggFunc       0      1      ptr(0x790700)
25    Next          0      9
26    Close         0      0
27    AggNext       0      37
28    AggGet        0      2
29    Integer       10     0      10
30    Ge            1      27
31    AggGet        0      0
32    AggGet        0      1
33    AggGet        0      2
34    Add           0      0
```

```
35    Callback      2      0
36    Goto          0      27
37    Noop          0      0
38    Halt          0      0
```

The code generated in this last example is the same as the previous except for the addition of two conditional jumps used to implement the extra WHERE and HAVING clauses. The WHERE clause is implemented by instructions 9 through 11 in the query loop. The HAVING clause is implemented by instruction 28 through 30 in the output loop.

Using SELECT Statements As Terms In An Expression

The very name "Structured Query Language" tells us that SQL should support nested queries. And, in fact, two different kinds of nesting are supported. Any SELECT statement that returns a single-row, single-column result can be used as a term in an expression of another SELECT statement. And, a SELECT statement that returns a single-column, multi-row result can be used as the right-hand operand of the IN and NOT IN operators. We will begin this section with an example of the first kind of nesting, where a single-row, single-column SELECT is used as a term in an expression of another SELECT. Here is our example:

```
SELECT * FROM examp
WHERE two=(SELECT three FROM examp2
            WHERE four=5);
```

The way SQLite deals with this is to first run the inner SELECT (the one against examp2) and store its result in a private memory cell. SQLite then substitutes the value of this private memory cell for the inner SELECT when it evaluates the outer SELECT. The code looks like this:

```
addr  opcode        p1     p2     p3
----  ------------  -----  -----  ----------------------------------
0     String        0      0
1     MemStore      0      1
2     Integer       0      0
3     OpenRead      1      5      examp2
4     VerifyCookie  0      909
5     Rewind        1      13
6     Column        1      1
7     Integer       5      0      5
8     Ne            1      12
9     Column        1      0
10    MemStore      0      1
11    Goto          0      13
12    Next          1      6
```

```
13    Close         1      0
14    ColumnName    0      0      one
15    ColumnName    1      0      two
16    Integer       0      0
17    OpenRead      0      3      examp
18    Rewind        0      26
19    Column        0      1
20    MemLoad       0      0
21    Eq            1      25
22    Column        0      0
23    Column        0      1
24    Callback      2      0
25    Next          0      19
26    Close         0      0
27    Halt          0      0
```

The private memory cell is initialized to NULL by the first two instructions. Instructions 2 through 13 implement the inner SELECT statement against the examp2 table. Notice that instead of sending the result to a callback or storing the result on a sorter, the result of the query is pushed into the memory cell by instruction 10 and the loop is abandoned by the jump at instruction 11. The jump at instruction at 11 is vestigial and never executes.

The outer SELECT is implemented by instructions 14 through 25. In particular, the WHERE clause that contains the nested select is implemented by instructions 19 through 21. You can see that the result of the inner select is loaded onto the stack by instruction 20 and used by the conditional jump at 21.

When the result of a sub-select is a scalar, a single private memory cell can be used, as shown in the previous example. But when the result of a sub-select is a vector, such as when the sub-select is the right-hand operand of IN or NOT IN, a different approach is needed. In this case, the result of the sub-select is stored in a transient table and the contents of that table are tested using the Found or NotFound operators. Consider this example:

```
SELECT * FROM examp
WHERE two IN (SELECT three FROM examp2);
```

The code generated to implement this last query is as follows:

```
addr  opcode        p1     p2     p3
----  ------------  -----  -----  ----------------------------------
0     OpenTemp      1      1
1     Integer       0      0
2     OpenRead      2      5      examp2
3     VerifyCookie  0      909
4     Rewind        2      10
5     Column        2      0
```

```
6     IsNull        -1     9
7     String        0      0
8     PutStrKey     1      0
9     Next          2      5
10    Close         2      0
11    ColumnName    0      0      one
12    ColumnName    1      0      two
13    Integer       0      0
14    OpenRead      0      3      examp
15    Rewind        0      25
16    Column        0      1
17    NotNull       -1     20
18    Pop           1      0
19    Goto          0      24
20    NotFound      1      24
21    Column        0      0
22    Column        0      1
23    Callback      2      0
24    Next          0      16
25    Close         0      0
26    Halt          0      0
```

The transient table in which the results of the inner SELECT are stored is created by the OpenTemp instruction at 0. This opcode is used for tables that exist for the duration of a single SQL statement only. The transient cursor is always opened read/write even if the main database is read-only. The transient table is deleted automatically when the cursor is closed. The P2 value of 1 means the cursor points to a BTree index, which has no data but can have an arbitrary key.

The inner SELECT statement is implemented by instructions 1 through 10. All this code does is make an entry in the temporary table for each row of the examp2 table with a non-NULL value for the "three" column. The key for each temporary table entry is the "three" column of examp2 and the data is an empty string since it is never used.

The outer SELECT is implemented by instructions 11 through 25. In particular, the WHERE clause containing the IN operator is implemented by instructions at 16, 17, and 20. Instruction 16 pushes the value of the "two" column for the current row onto the stack and instruction 17 checks to see that it is non-NULL. If this is successful, execution jumps to 20, where it tests to see if top of the stack matches any key in the temporary table. The rest of the code is the same as what has been shown before.

Compound SELECT Statements

SQLite also allows two or more SELECT statements to be joined as peers using operators UNION, UNION ALL, INTERSECT, and EXCEPT. These compound select

statements are implemented using transient tables. The implementation is slightly different for each operator, but the basic ideas are the same. For an example we will use the EXCEPT operator.

```
SELECT two FROM examp
EXCEPT
SELECT four FROM examp2;
```

The result of this last example should be every unique value of the "two" column in the examp table, except any value that is in the "four" column of examp2 is removed. The code to implement this query is as follows:

```
addr  opcode        p1     p2     p3
----  ------------  -----  -----  ----------------------------------
0     OpenTemp      0      1
1     KeyAsData     0      1
2     Integer       0      0
3     OpenRead      1      3      examp
4     VerifyCookie  0      909
5     Rewind        1      11
6     Column        1      1
7     MakeRecord    1      0
8     String        0      0
9     PutStrKey     0      0
10    Next          1      6
11    Close         1      0
12    Integer       0      0
13    OpenRead      2      5      examp2
14    Rewind        2      20
15    Column        2      1
16    MakeRecord    1      0
17    NotFound      0      19
18    Delete        0      0
19    Next          2      15
20    Close         2      0
21    ColumnName    0      0      four
22    Rewind        0      26
23    Column        0      0
24    Callback      1      0
25    Next          0      23
26    Close         0      0
27    Halt          0      0
```

The transient table in which the result is built is created by instruction 0. Three loops then follow. The loop at instructions 5 through 10 implements the first SELECT statement. The second SELECT statement is implemented by the loop at instructions

14 through 19. Finally, a loop at instructions 22 through 25 reads the transient table and invokes the callback once for each row in the result.

Instruction 1 is of particular importance in this example. Normally, the Column instruction extracts the value of a column from a larger record in the data of an SQLite file entry. Instruction 1 sets a flag on the transient table so that Column will instead treat the key of the SQLite file entry as if it were data and extract column information from the key.

Here is what is going to happen: The first SELECT statement will construct rows of the result and save each row as the key of an entry in the transient table. The data for each entry in the transient table is a never used so we fill it in with an empty string. The second SELECT statement also constructs rows, but the rows constructed by the second SELECT are removed from the transient table. That is why we want the rows to be stored in the key of the SQLite file instead of in the data -- so they can be easily located and deleted.

Let's look more closely at what is happening here. The first SELECT is implemented by the loop at instructions 5 through 10. Instruction 5 initializes the loop by rewinding its cursor. Instruction 6 extracts the value of the "two" column from "examp" and instruction 7 converts this into a row. Instruction 8 pushes an empty string onto the stack. Finally, instruction 9 writes the row into the temporary table. But remember, the PutStrKey opcode uses the top of the stack as the record data and the next on stack as the key. For an INSERT statement, the row generated by the MakeRecord opcode is the record data and the record key is an integer created by the NewRecno opcode. But here the roles are reversed and the row created by MakeRecord is the record key and the record data is just an empty string.

The second SELECT is implemented by instructions 14 through 19. Instruction 14 initializes the loop by rewinding its cursor. A new result row is created from the "four" column of table "examp2" by instructions 15 and 16. But instead of using PutStrKey to write this new row into the temporary table, we instead call Delete to remove it from the temporary table if it exists.

The result of the compound select is sent to the callback routine by the loop at instructions 22 through 25. There is nothing new or remarkable about this loop, except for the fact that the Column instruction at 23 will be extracting a column out of the record key rather than the record data.

Summary

This article has reviewed all of the major techniques used by SQLite's VDBE to implement SQL statements. What has not been shown is that most of these techniques can be used in combination to generate code for an appropriately complex query statement. For example, we have shown how sorting is accomplished on a simple query and we have shown how to implement a compound query. But we did not give an example of sorting in a compound query. This is because sorting a

compound query does not introduce any new concepts: it merely combines two previous ideas (sorting and compounding) in the same VDBE program.

For additional information on how the SQLite library functions, the reader is directed to look at the SQLite source code directly. If you understand the material in this article, you should not have much difficulty in following the sources. Serious students of the internals of SQLite will probably also want to make a careful study of the VDBE opcodes as documented here. Most of the opcode documentation is extracted from comments in the source code using a script so you can also get information about the various opcodes directly from the **vdbe.c** source file. If you have successfully read this far, you should have little difficulty understanding the rest.

If you find errors in either the documentation or the code, feel free to fix them and/or contact the author at drh@hwaci.com. Your bug fixes or suggestions are always welcomed.

This page last modified on 2022-01-08 05:02:57 UTC