Inside SQLite

Table Of Contents

Table of Contents

Inside SQLite

by Sibsankar Haldar

Copyright

Chapter 1. Overview

Many database management systems (DBMSs) have been developed over the past several decades. DB-2, Informix, Ingres, MySQL, Oracle, SQL Server, Sybase are a few that are commercially successful. SQLite is a recent addition to the DBMS family, and it is also quite successful commercially. It has made a long journey since its debut on May 29, 2000.[*] It is an SQL-based relational DBMS (RDBMS) with the following characteristics:

[*] SQLite received the 2005 OSCON Google and O'Reilly Integrator category award.

Zero configuration

> There is no separate install or set up procedure to initialize SQLite before using it. There is no configuration file. Databases need minimal or no administration. You can download the SQLite source code from its homepage http://www.sqlite.org, compile it using your favorite C compiler, and start using the compiled library.

Embeddable

There is no need to maintain a separate server process dedicated to SQLite. The library is embeddable in your own applications.

Application interface

SQLite provides an SQL environment for C applications to manipulate databases. It is a call-level interface library for dynamic SQL; users can assemble SQL statements on the fly and pass them on to the interface for execution. (There are no other means, except by SQL, to manipulate databases.) There are no special preprocessing and compilation requirements for applications; a normal C compiler will do the work.

Transactional support

SQLite supports the core transactional properties, namely atomicity, consistency, isolation, and durability (ACID). No actions are required from database administrators upon a system crash or power failure to recover databases. When SQLite reads a database, it automatically performs necessary recovery actions on the database in a user-transparent manner.

Thread-safe

SQLite is a thread-safe library, and many threads in an application process can access the same or different databases concurrently. SQLite takes care of database-level thread concurrency.

Lightweight

SQLite has a small footprint of about 250 KB, and the footprint can be reduced further by disabling some advanced features when you compile it from the source code. SQLite runs on Linux, Windows, Mac OS X, OpenBSD, and a few other operating systems.

Customizable

SQLite provides a good framework in which you can define and use custom-made SQL functions, aggregate functions, and collating sequences. It also supports UTF-8 and UTF-16 standards-based encoding for Unicode text.

Cross-platform

SQLite lets you move database files between platforms. For example, you can create a database on a Linux x86 machine, and use the same database (by making a copy) on an ARM platform without any alterations. The database behaves identically on all supported platforms.

SQLite is different from most other modern SQL databases in the sense that its primary design goal is to be simple. SQLite strives to be simple, even if it leads to occasional inefficient implementations of some features. It is simple to maintain, customize, operate, administer, and embed in C applications. It uses simple techniques to implement ACID properties.

SQLite supports a large subset of SQL-92 data definition and manipulation features, and some SQLite specific commands. You can create tables, indexes, triggers, and views using standard data definition SQL constructs. You can manipulate stored information using INSERT, DELETE, UPDATE, and SELECT SQL constructs. The following is a list of additional features supported as of the SQLite 3.3.6 release (the latest set of supported features can be obtained from the SQLite homepage):

- Partial support for ALTER TABLE (rename table, add column)
- UNIQUE, NOT NULL, CHECK constraints
- Subqueries, including correlated subqueries, INNER JOIN, LEFT OUTER JOIN, NATURAL JOIN, UNION, UNION ALL, INTERSECT, EXCEPT
- Transactional commands: BEGIN, COMMIT, ROLLBACK
- SQLite commands, including reindex, attach, detach, pragma

The following SQL-92 features are not yet supported as of the SQLite 3.3.6 release:

- Foreign key constraints (parsed but not enforced)
- Many ALTER TABLE features
- Some TRIGGER-related features
- RIGHT and FULL OUTER JOIN
- Updating a VIEW
- GRANT and REVOKE

SQLite stores an entire database in a single, ordinary native file that can reside anywhere in a directory of the native filesystem. Any user who has permission to read the file can read anything from the database. A user who has write permission on the file and the container directory can change anything in the database. SQLite uses a separate journal file to save

transaction recovery information that is used in the event of transaction aborts or system failures. The SQLite *attach* command helps a transaction work on multiple databases simultaneously. Such transactions are also ACID-compliant. Pragma commands let you alter the behavior of the SQLite library.

SQLite allows multiple applications to access the same database concurrently. However, it supports a limited form of concurrency among transactions. It allows any number of concurrent read-transactions on a database, but only one exclusive write-transaction.

SQLite is a successful RDBMS. It has been widely used in low-to-medium tier database applications such as web site services (SQLite works fine with up to 100,000 evenly distributed hits a day; the SQLite development team has demonstrated that SQLite may even withstand 1,000,000 hits a day), cell phones, PDAs, set-top boxes, standalone appliances. You could even use it in teaching SQL to students in a beginner database course. The source code itself is well documented and commented, and you can use it in advanced database management courses, or in projects as a reference technology. It is available freely, and has no licensing complications because it is in the public domain.

1.1. Sample Applications

In this section I will present some simple applications illustrating various features of SQLite. The applications are presented in the following subsections.

Let's begin our exploration of SQLite land by studying a very simple application. The following example presents a typical SQLite application. It is a C program that invokes SQLite APIs to work with an SQLite database file:

Here is a typical SQLite application:

Code View: Scroll / Show All

```c
#include <stdio.h>
#include "sqlite3.h"

int main(void)
{
  sqlite3*     db = 0;
  sqlite3_stmt* stmt = 0;
  int retcode;
  retcode = sqlite3_open("MyDB", &db); /* Open a database named MyDB */
  if (retcode != SQLITE_OK){
    sqlite3_close(db);
    fprintf(stderr, "Could not open MyDB\n");
    return retcode;
```

```
  }
  retcode = sqlite3_prepare(db, "select SID from Students order by SID",
                     -1, &stmt, 0);
  if (retcode != SQLITE_OK){
    sqlite3_close(db);
    fprintf(stderr, "Could not execute SELECT\n");
    return retcode;
  }
  while (sqlite3_step(stmt) == SQLITE_ROW){
    int i = sqlite3_column_int(stmt, 0);
    printf("SID = %d\n", i);
  }
  sqlite3_finalize(stmt);
  sqlite3_close(db);
  return SQLITE_OK;
}
```

You may compile the above example application and execute it. The sample output shown throughout this document was generated on a Linux machine, but these examples will work on other platforms that SQLite runs on.

These examples assume that you have already prepared the *sqlite3* executable, the *libsqlite3.so* (*sqlite3.dll* on Windows and *libsqlite3.dylib* for Mac OS X) shared library, and the *sqlite3.h* interface definition file. You can obtain these in source or binary form from http://www.sqlite.org. You may find it easier to work with these examples if you put all three (*sqlite3*, the shared library, and *sqlite3.h*) in the same directory as the examples.

For example, suppose you are on a Linux system and that you've saved the *app1.c* sample program in the same directory as *libsqlite3.so*, *sqlite3*, and *sqlite3.h*. You can compile the file by executing this command:

```
gcc app1.c -o ./app1 -lsqlite3 -L.
```

It will produce a binary named `app1` in the current working directory. You may execute the binary to see the output.

NOTE

Both the SQLite source code and the application must be compiled with the same compiler.

If you've installed SQLite as a package, or if your operating system came with it preinstalled, you may need to use a different set of compiler arguments. For example, on Ubuntu, you can install SQLite with the command `sudo aptitude install sqlite3 libsqlite3-dev`, and you can compile the example application with the command `cc app1.c -o ./app1 -lsqlite3`.

Because SQLite is included with recent versions of Mac OS X, this same compilation command works there as well.

The application opens the MyDB database file in the current working directory. The database needs at least one table, named Students; this table must have at least one integer column named SID. In the next example application, you will learn how to create new tables in databases, and how to insert rows (also called *tuples* and *records*) in tables, but for the moment, you can create and populate the table with these commands:

```
sqlite3 MyDB "create table students (SID integer)"
sqlite3 MyDB "insert into students values (200)"
sqlite3 MyDB "insert into students values (100)"
sqlite3 MyDB "insert into students values (300)"
```

If you run the `app1` now (to pull in the SQLite library, you may need to include your working directory name in the LD_LIBRARY_PATH environment variable on Linux systems), you will see the following output:

```
SID = 100
SID = 200
SID = 300
```

NOTE

On Linux, Unix, and Mac OS X, you may need to prefix `app1` with `./` when you type its name at the command prompt, as in:

```
./app1
```

The application first prepares, then executes the SQL statement: `select SID from Students order by SID`. It then steps through the resulting rowset, fetches SID values one by one, and prints the values. Finally, it closes the database.

SQLite is a *call-level interface library* that is embedded in applications. The library implements all SQLite APIs as C functions. All API function names are prefixed with `sqlite3_` and their signatures are declared in *sqlite3.h*. A few of them are used in the example application, namely `sqlite3_open`, `sqlite3_prepare`, `sqlite3_step`, `sqlite3_column_int`, `sqlite3_finalize`, and `sqlite3_close`. The application also uses mnemonic constants, namely `SQLITE_OK` and

SQLITE_ROW, for comparing values returned by the APIs. The mnemonics are defined in *sqlite3.h*.

The following sections discuss some of the key functions in the SQLite API.

1.1.1.1. sqlite3_open

By executing the sqlite3_open function, an application opens a new connection to a database file via the SQLite library. (The application may have other open connections to the same or different databases. SQLite treats the connections distinctly, and they are independent of one another as far as SQLite is concerned.) SQLite automatically creates the database file if the file does not already exist.

NOTE

When opening or creating a file, SQLite follows a lazy approach: the actual opening or creation is deferred until the file is accessed for reading.

The sqlite3_open function returns a connection handle (a pointer to an object of type sqlite3) via a formal parameter (db, in the preceding example), and the handle is used to apply further operations on the database connection (for this open connection). The handle represents the complete state of this connection.

1.1.1.2. sqlite3_prepare

The sqlite3_prepare function compiles an SQL statement, and produces an equivalent internal object. This object is commonly referred to as a *prepared statement* in database literature, and is implemented as a *bytecode program* in SQLite. A bytecode program is an abstract representation of an SQL statement that is run on a virtual machine or an interpreter. For more information, see the later section, Bytecode Programming Language. I use the terms bytecode program and prepared statement interchangeably in this book.

The sqlite3_prepare function returns a statement handle (a pointer to an object of type sqlite3_stmt) via a formal parameter (stmt in the preceding example), and the handle is used to apply further operations to manipulate the prepared statement. In the example program, I prepared the select SID from Students order by SID statement as the stmt handle. This handle acts like an open cursor and is used to obtain the resulting rowset that the SELECT statement returns, one row at a time.

1.1.1.3. sqlite3_step

The sqlite3_step function executes the bytecode program until it hits a break point (because it has computed a new row), or until it halts (there are no more rows). In the former case, it returns SQLITE_ROW, and in the latter case, SQLITE_DONE. For SQL statements that don't return rows (such as UPDATE, INSERT, DELETE, and CREATE), it always returns SQLITE_DONE because there are no rows to process. The step function moves the position of the cursor for

the results of a SELECT statement. Initially, the cursor points before the first row of the output rowset. Every execution of the `step` function moves the cursor pointer to the next row of the rowset. The cursor moves only in the forward direction.

1.1.1.4. sqlite3_column_int

If the `step` function returns `SQLITE_ROW`, you can retrieve the value of each column (also known as attribute or field) by executing the `sqlite3_column_*` API functions. The impedance (data type) mismatch between SQL/SQLite and the C language is handled automatically: the APIs convert data between the two languages and from storage types to requested types. In the example application, each output row is an integer value, and we read the value of SID by executing the `sqlite3_column_int` function that returns integer values.

1.1.1.5. sqlite3_finalize

The `sqlite3_finalize` function destroys the prepared statement. That is, it erases the bytecode program, and frees all resources allocated to the statement handle. The handle becomes invalid.

1.1.1.6. sqlite3_close

The `sqlite3_close` function closes the database connection, and frees all resources allocated to the connection. The connection handle becomes invalid.

1.1.1.7. Other useful functions

The other widely used APIs are `sqlite3_bind_*` and `sqlite3_reset`. In an SQL statement string (input to `sqlite3_prepare`), one or more literal values can be replaced by the SQL parameter marker `?` (or `?NNN`, `:AAA`, `@AAA`, or `$AAA` where `NNN` is a number and `AAA` is an identifier). They become input parameters to the prepared statement. The values of these parameters can be set using the `bind` API functions. If no value is bound to a parameter, either the default value is taken, or SQL NULL is taken if no default is declared in the schema. The `reset` API function resets a statement handle (i.e., the prepared statement) back to its initial state with one exception: all parameters that had values bound to them retain their values. The statement becomes ready to be reexecuted by the application, and reuses these values in the reexecution. However, the application may replace these values by new ones executing the `bind` APIs once again before it starts the reexecution.

1.1.1.8. Return values

All API functions return zero or positive integer values. SQLite recommends using mnemonics for return value checks. The return value `SQLITE_OK` indicates a success; `SQLITE_ROW` indicates that the `sqlite3_step` function has found a new row in the rowset that the SELECT statement returned; `SQLITE_DONE` indicates that the statement execution is complete.

The next example presents another SQLite application that can be run from a command line to manipulate databases interactively. It takes two arguments: the first one is a database file name, and the second one an SQL statement. It first opens the database file, then applies the statement to the database by executing the sqlite3_exec API function, and finally closes the database file. The function executes the statement directly, without the need to go through the prepare and step API functions, as was done in the previous example. In case the statement produces output, the exec function executes the callback function for each output row. The user must have read permission on the given database file, and depending on the query type, she may need to have a write permission on the file and the directory it's contained in. Here is a command-line based application:

Code View: Scroll / <u>Show All</u>

```c
#include <stdio.h>
#include "sqlite3.h"

static int callback(void *NotUsed, int argc, char **argv, char **colName)
{
  // Loop over each column in the current row
  int i;
  for (i = 0; i < argc; i++){
    printf("%s = %s\n", colName[i], argv[i] ? argv[i] : "NULL");
  }
  return 0;
}

int main(int argc, char **argv)
{
  sqlite3* db = 0;
  char* zErrMsg = 0;
  int rc;

  if (argc != 3){
    fprintf(stderr, "Usage: %s DATABASE-FILE-NAME SQL-STATEMENT\n", argv[0]);
    return -1;
  }
  rc = sqlite3_open(argv[1], &db);
  if (rc != SQLITE_OK){
    fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
    sqlite3_close(db);
    return -2;
  }
  rc = sqlite3_exec(db, argv[2], callback, 0, &zErrMsg);
  if (rc != SQLITE_OK){
    fprintf(stderr, "SQL error: %s\n", zErrMsg);
  }
```

```
    sqlite3_close(db);

    return rc;
}
```

Compile the preceding application code into an executable, such as `app2`. You can now issue SQL statements that operate on a database. Suppose you are working on the same MyDB database in the current working directory. By executing the following command lines, you can insert new rows in the Students table:

```
./app2 MyDB "insert into Students values(100)"
./app2 MyDB "insert into Students values(10)"
./app2 MyDB "insert into Students values(1000)"
```

If you run the previous application (`app1`) now, you will see the following output:

```
SID = 10
SID = 100
SID = 100
SID = 200
SID = 300
SID = 1000
```

You can also create new tables in databases; for example, `./app2 MyDBExtn "create table Courses(name varchar, SID integer)"` creates the `Courses` table in a new MyDBExtn database in the current working directory.

NOTE

SQLite has an interactive command-line program (*sqlite3*), shown earlier, which you can use to issue SQL commands. You can download a precompiled binary version of it from the SQLite site, or compile it from source. This `app2` example is essentially a bare-bones implementation of *sqlite3*.

SQLite is a thread-safe library. Consequently, many threads in an application can access the same or different databases concurrently.

NOTE

In order to be thread-safe, the SQLite source code must be compiled with the THREADSAFE preprocessor macro set to 1 (enable this by passing `--enable-threadsafe` to the *configure* script when you prepare to compile SQLite). In the default setting, the multithreading feature is turned off for Linux, but turned on for Windows. There is no way you can query the SQLite library to find out whether it has the feature.

This example uses the pthreads library, which is not included with Windows. On Windows, you have two choices: use the tools and compilers from Cygwin (http://www.cygwin.com), or download pthreads for Windows from http://sourceware.org/pthreads-win32/.

Here is a typical multithreaded application:

Code View: Scroll / <u>Show All</u>

```c
#include <stdio.h>
#include <pthread.h>
#include "sqlite3.h"

void* myInsert(void* arg)
{
    sqlite3*     db = 0;
    sqlite3_stmt* stmt = 0;
    int val = (int)arg;
    char SQL[100];
    int rc;
    rc = sqlite3_open("MyDB", &db); /* Open a database named MyDB */
    if (rc != SQLITE_OK) {
      fprintf(stderr, "Thread[%d] fails to open the database\n",  val);
      goto errorRet;
    }

    /* Create the SQL string. If you were using string values,
       you would need to use sqlite3_prepare() and sqlite3_bind_*
       to avoid an SQL injection vulnerability. However %d
       guarantees an integer value, so this use of sprintf is
       safe.
     */
    sprintf(SQL, "insert into Students values(%d)", val);

    /* Prepare the insert statement */
    rc = sqlite3_prepare(db, SQL, -1, &stmt, 0);
    if (rc != SQLITE_OK) {
      fprintf(stderr, "Thread[%d] fails to prepare SQL: %s ->
return code %d\n", val, SQL, rc);
      goto errorRet;
    }
```

```
    rc = sqlite3_step(stmt);

    if (rc != SQLITE_DONE) {
      fprintf(stderr,
        "Thread[%d] fails to execute SQL: %s -> return code %d\n", val, SQL, rc);
    }
    else {
      printf("Thread[%d] successfully executes SQL: %s\n", val,
        SQL);
    }
    errorRet:
      sqlite3_close(db);
      return (void*)rc;
}
int main(void)
{
  pthread_t t[10];
  int i;
  for (i=0; i < 10; i++)
    pthread_create(&t[i], 0, myInsert, (void*)i); /* Pass the value of i */

  /* wait for all threads to finish */
  for (i=0; i<10; i++)  pthread_join(t[i], 0);
  return 0;
}
```

> This application may not work "out of the box" on Windows and Mac OS X. You may need to recompile SQLite with threading support, and/or obtain the pthread libraries to make the application work on those platforms. Mac OS X includes pthreads, and you can obtain pthreads for Windows at http://sourceware.org/pthreads-win32/.

The application creates 10 threads, and each of them tries to insert one row in the Students table in the same MyDB database. SQLite implements a lock-based concurrency scheme, so some or all of the INSERT statements may fail due to lock conflict. The application does not need to worry about concurrency control and database consistency issues; it cannot corrupt the database. SQLite takes care of concurrency control and consistency issues. However, you will need to check for failures, and handle them appropriately within the code (for example, you might retry the command that failed, or inform the user that it failed and let her decide what to do next).

Each thread needs to connect to its database, and needs to create its own SQLite (connection and statement) handles. SQLite does not recommend sharing handles across threads. Even though sharing handles between threads may appear to work, there is no guarantee that you'll get the expected results. In fact, the SQLite library may break and dump core in some versions of Linux. Also, under Unix/Linux systems, you should not attempt to preserve connection handles across a `fork` system call into the child process. Problems such as database corruption or an application crash may arise if you do this.

NOTE

The restriction against sharing a database connection between threads is somewhat relaxed in SQLite 3.3.1 and subsequent versions. Threads can use a database connection safely in mutual exclusion. What this means is that you can switch a connection from one thread to another as long as the former thread does not hold any native file locks on the connection. You can safely assume that no locks are held if the thread has no pending transaction, and if it has reset or finalized all statements on the connection.

Here is a typical SQLite application that uses two databases:

Code View: Scroll / <u>Show All</u>

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

int main(void)
{
   sqlite3* db = 0;

   sqlite3_open("MyDB", &db); /* Open a database named MyDB */
   sqlite3_exec(db, "attach database MyDBExtn as DB1", 0, 0, 0);
   sqlite3_exec(db, "begin", 0, 0, 0);
   sqlite3_exec(db, "insert into Students values(2000)", 0, 0, 0);
   sqlite3_exec(db, "insert into Courses values('SQLite Database', 2000)", 0, 0, 0);
   sqlite3_exec(db, "commit", 0, 0, 0);

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

I have simplified the code by not including error checks. The application opens the MyDB database, and then attaches the MyDBExtn database to the current connection. MyDB needs to have a Students(SID) table, and the MyDBExtn database needs a Courses(name, SID) table. The application opens a transaction by executing the `begin` command, inserts one row in Students and one in Courses inside the transaction, and finally commits the transaction by executing the `commit` command. INSERT statements do not need a callback function, and hence, I pass 0 as the callback parameter in `sqlite3_exec` calls in the example application.

NOTE

SQLite permits multiple SQL statements in a single `exec` API call; the same batch of commands can be executed by passing this sequence of statements in a single `exec` call: `begin; insert into Students values(2000); insert into Courses values('SQLite Database', 2000); commit`. If the batch contains SELECT statements, the same callback function is used to process the resulting rowsets.

A *catalog* is a system table that is created and maintained by SQLite itself. It stores meta information about the database. SQLite maintains one master catalog, named `sqlite_master`, in every database. The catalog stores schema information about tables, indexes, triggers, and views. You can query the master catalog (e.g., `select * from sqlite_master`), but cannot directly modify the catalog. There are other optional catalog tables. All catalog table names start with the `sqlite_` prefix, and the names are reserved by SQLite for internal use. You cannot create database objects (tables, views, indexes, triggers) with such names (in upper, lower or mixed case).

Section 1.1. Sample Applications

Chapter 1. Overview

Many database management systems (DBMSs) have been developed over the past several decades. DB-2, Informix, Ingres, MySQL, Oracle, SQL Server, Sybase are a few that are commercially successful. SQLite is a recent addition to the DBMS family, and it is also quite successful commercially. It has made a long journey since its debut on May 29, 2000.[*] It is an SQL-based relational DBMS (RDBMS) with the following characteristics:

[*] SQLite received the 2005 OSCON Google and O'Reilly Integrator category award.

Zero configuration

There is no separate install or set up procedure to initialize SQLite before using it. There is no configuration file. Databases need minimal or no administration. You

can download the SQLite source code from its homepage http://www.sqlite.org, compile it using your favorite C compiler, and start using the compiled library.

Embeddable

There is no need to maintain a separate server process dedicated to SQLite. The library is embeddable in your own applications.

Application interface

SQLite provides an SQL environment for C applications to manipulate databases. It is a call-level interface library for dynamic SQL; users can assemble SQL statements on the fly and pass them on to the interface for execution. (There are no other means, except by SQL, to manipulate databases.) There are no special preprocessing and compilation requirements for applications; a normal C compiler will do the work.

Transactional support

SQLite supports the core transactional properties, namely atomicity, consistency, isolation, and durability (ACID). No actions are required from database administrators upon a system crash or power failure to recover databases. When SQLite reads a database, it automatically performs necessary recovery actions on the database in a user-transparent manner.

Thread-safe

SQLite is a thread-safe library, and many threads in an application process can access the same or different databases concurrently. SQLite takes care of database-level thread concurrency.

Lightweight

SQLite has a small footprint of about 250 KB, and the footprint can be reduced further by disabling some advanced features when you compile it from the source

code. SQLite runs on Linux, Windows, Mac OS X, OpenBSD, and a few other operating systems.

Customizable

SQLite provides a good framework in which you can define and use custom-made SQL functions, aggregate functions, and collating sequences. It also supports UTF-8 and UTF-16 standards-based encoding for Unicode text.

Cross-platform

SQLite lets you move database files between platforms. For example, you can create a database on a Linux x86 machine, and use the same database (by making a copy) on an ARM platform without any alterations. The database behaves identically on all supported platforms.

SQLite is different from most other modern SQL databases in the sense that its primary design goal is to be simple. SQLite strives to be simple, even if it leads to occasional inefficient implementations of some features. It is simple to maintain, customize, operate, administer, and embed in C applications. It uses simple techniques to implement ACID properties.

SQLite supports a large subset of SQL-92 data definition and manipulation features, and some SQLite specific commands. You can create tables, indexes, triggers, and views using standard data definition SQL constructs. You can manipulate stored information using INSERT, DELETE, UPDATE, and SELECT SQL constructs. The following is a list of additional features supported as of the SQLite 3.3.6 release (the latest set of supported features can be obtained from the SQLite homepage):

- Partial support for ALTER TABLE (rename table, add column)
- UNIQUE, NOT NULL, CHECK constraints
- Subqueries, including correlated subqueries, INNER JOIN, LEFT OUTER JOIN, NATURAL JOIN, UNION, UNION ALL, INTERSECT, EXCEPT
- Transactional commands: BEGIN, COMMIT, ROLLBACK
- SQLite commands, including reindex, attach, detach, pragma

The following SQL-92 features are not yet supported as of the SQLite 3.3.6 release:

- Foreign key constraints (parsed but not enforced)
- Many ALTER TABLE features
- Some TRIGGER-related features
- RIGHT and FULL OUTER JOIN
- Updating a VIEW

- GRANT and REVOKE

SQLite stores an entire database in a single, ordinary native file that can reside anywhere in a directory of the native filesystem. Any user who has permission to read the file can read anything from the database. A user who has write permission on the file and the container directory can change anything in the database. SQLite uses a separate journal file to save transaction recovery information that is used in the event of transaction aborts or system failures. The SQLite *attach* command helps a transaction work on multiple databases simultaneously. Such transactions are also ACID-compliant. Pragma commands let you alter the behavior of the SQLite library.

SQLite allows multiple applications to access the same database concurrently. However, it supports a limited form of concurrency among transactions. It allows any number of concurrent read-transactions on a database, but only one exclusive write-transaction.

SQLite is a successful RDBMS. It has been widely used in low-to-medium tier database applications such as web site services (SQLite works fine with up to 100,000 evenly distributed hits a day; the SQLite development team has demonstrated that SQLite may even withstand 1,000,000 hits a day), cell phones, PDAs, set-top boxes, standalone appliances. You could even use it in teaching SQL to students in a beginner database course. The source code itself is well documented and commented, and you can use it in advanced database management courses, or in projects as a reference technology. It is available freely, and has no licensing complications because it is in the public domain.

1.1. Sample Applications

In this section I will present some simple applications illustrating various features of SQLite. The applications are presented in the following subsections.

Let's begin our exploration of SQLite land by studying a very simple application. The following example presents a typical SQLite application. It is a C program that invokes SQLite APIs to work with an SQLite database file:

Here is a typical SQLite application:

Code View: Scroll / Show All
```c
#include <stdio.h>
#include "sqlite3.h"

int main(void)
{
  sqlite3*     db = 0;
  sqlite3_stmt* stmt = 0;
```

```
    int retcode;
    retcode = sqlite3_open("MyDB", &db); /* Open a database named MyDB */
    if (retcode != SQLITE_OK){
        sqlite3_close(db);
        fprintf(stderr, "Could not open MyDB\n");
        return retcode;
    }
    retcode = sqlite3_prepare(db, "select SID from Students order by SID",
                          -1, &stmt, 0);
    if (retcode != SQLITE_OK){
        sqlite3_close(db);
        fprintf(stderr, "Could not execute SELECT\n");
        return retcode;
    }
    while (sqlite3_step(stmt) == SQLITE_ROW){
        int i = sqlite3_column_int(stmt, 0);
        printf("SID = %d\n", i);
    }
    sqlite3_finalize(stmt);
    sqlite3_close(db);
    return SQLITE_OK;
}
```

You may compile the above example application and execute it. The sample output shown throughout this document was generated on a Linux machine, but these examples will work on other platforms that SQLite runs on.

These examples assume that you have already prepared the *sqlite3* executable, the *libsqlite3.so* (*sqlite3.dll* on Windows and *libsqlite3.dylib* for Mac OS X) shared library, and the *sqlite3.h* interface definition file. You can obtain these in source or binary form from http://www.sqlite.org. You may find it easier to work with these examples if you put all three (*sqlite3*, the shared library, and *sqlite3.h*) in the same directory as the examples.

For example, suppose you are on a Linux system and that you've saved the *app1.c* sample program in the same directory as *libsqlite3.so*, *sqlite3*, and *sqlite3.h*. You can compile the file by executing this command:

```
gcc app1.c -o ./app1 -lsqlite3 -L.
```

It will produce a binary named `app1` in the current working directory. You may execute the binary to see the output.

NOTE

Both the SQLite source code and the application must be compiled with the same compiler.

If you've installed SQLite as a package, or if your operating system came with it preinstalled, you may need to use a different set of compiler arguments. For example, on Ubuntu, you can install SQLite with the command `sudo aptitude install sqlite3 libsqlite3-dev`, and you can compile the example application with the command `cc app1.c -o ./app1 -lsqlite3`.

Because SQLite is included with recent versions of Mac OS X, this same compilation command works there as well.

The application opens the MyDB database file in the current working directory. The database needs at least one table, named Students; this table must have at least one integer column named SID. In the next example application, you will learn how to create new tables in databases, and how to insert rows (also called *tuples* and *records*) in tables, but for the moment, you can create and populate the table with these commands:

```
sqlite3 MyDB "create table students (SID integer)"
sqlite3 MyDB "insert into students values (200)"
sqlite3 MyDB "insert into students values (100)"
sqlite3 MyDB "insert into students values (300)"
```

If you run the `app1` now (to pull in the SQLite library, you may need to include your working directory name in the LD_LIBRARY_PATH environment variable on Linux systems), you will see the following output:

```
SID = 100
SID = 200
SID = 300
```

NOTE

On Linux, Unix, and Mac OS X, you may need to prefix `app1` with `./` when you type its name at the command prompt, as in:

```
./app1
```

The application first prepares, then executes the SQL statement: `select SID from Students order by SID`. It then steps through the resulting rowset, fetches SID values one by one, and prints the values. Finally, it closes the database.

SQLite is a *call-level interface library* that is embedded in applications. The library implements all SQLite APIs as C functions. All API function names are prefixed with `sqlite3_` and their signatures are declared in *sqlite3.h*. A few of them are used in the example application, namely `sqlite3_open`, `sqlite3_prepare`, `sqlite3_step`, `sqlite3_column_int`, `sqlite3_finalize`, and `sqlite3_close`. The application also uses mnemonic constants, namely `SQLITE_OK` and `SQLITE_ROW`, for comparing values returned by the APIs. The mnemonics are defined in *sqlite3.h*.

The following sections discuss some of the key functions in the SQLite API.

1.1.1.1. sqlite3_open

By executing the `sqlite3_open` function, an application opens a new connection to a database file via the SQLite library. (The application may have other open connections to the same or different databases. SQLite treats the connections distinctly, and they are independent of one another as far as SQLite is concerned.) SQLite automatically creates the database file if the file does not already exist.

NOTE

When opening or creating a file, SQLite follows a lazy approach: the actual opening or creation is deferred until the file is accessed for reading.

The `sqlite3_open` function returns a connection handle (a pointer to an object of type `sqlite3`) via a formal parameter (`db`, in the preceding example), and the handle is used to apply further operations on the database connection (for this open connection). The handle represents the complete state of this connection.

1.1.1.2. sqlite3_prepare

The `sqlite3_prepare` function compiles an SQL statement, and produces an equivalent internal object. This object is commonly referred to as a *prepared statement* in database literature, and is implemented as a *bytecode program* in SQLite. A bytecode program is an abstract representation of an SQL statement that is run on a virtual machine or an interpreter. For more information, see the later section, Bytecode Programming Language. I use the terms bytecode program and prepared statement interchangeably in this book.

The `sqlite3_prepare` function returns a statement handle (a pointer to an object of type `sqlite3_stmt`) via a formal parameter (`stmt` in the preceding example), and the handle is used to apply further operations to manipulate the prepared statement. In the example program, I prepared the `select SID from Students order by SID` statement as the `stmt`

handle. This handle acts like an open cursor and is used to obtain the resulting rowset that the SELECT statement returns, one row at a time.

1.1.1.3. `sqlite3_step`

The `sqlite3_step` function executes the bytecode program until it hits a break point (because it has computed a new row), or until it halts (there are no more rows). In the former case, it returns `SQLITE_ROW`, and in the latter case, `SQLITE_DONE`. For SQL statements that don't return rows (such as UPDATE, INSERT, DELETE, and CREATE), it always returns `SQLITE_DONE` because there are no rows to process. The `step` function moves the position of the cursor for the results of a SELECT statement. Initially, the cursor points before the first row of the output rowset. Every execution of the `step` function moves the cursor pointer to the next row of the rowset. The cursor moves only in the forward direction.

1.1.1.4. `sqlite3_column_int`

If the `step` function returns `SQLITE_ROW`, you can retrieve the value of each column (also known as attribute or field) by executing the `sqlite3_column_*` API functions. The impedance (data type) mismatch between SQL/SQLite and the C language is handled automatically: the APIs convert data between the two languages and from storage types to requested types. In the example application, each output row is an integer value, and we read the value of SID by executing the `sqlite3_column_int` function that returns integer values.

1.1.1.5. `sqlite3_finalize`

The `sqlite3_finalize` function destroys the prepared statement. That is, it erases the bytecode program, and frees all resources allocated to the statement handle. The handle becomes invalid.

1.1.1.6. `sqlite3_close`

The `sqlite3_close` function closes the database connection, and frees all resources allocated to the connection. The connection handle becomes invalid.

1.1.1.7. Other useful functions

The other widely used APIs are `sqlite3_bind_*` and `sqlite3_reset`. In an SQL statement string (input to `sqlite3_prepare`), one or more literal values can be replaced by the SQL parameter marker `?` (or `?NNN`, `:AAA`, `@AAA`, or `$AAA` where `NNN` is a number and `AAA` is an identifier). They become input parameters to the prepared statement. The values of these parameters can be set using the `bind` API functions. If no value is bound to a parameter, either the default value is taken, or SQL NULL is taken if no default is declared in the schema. The `reset` API function resets a statement handle (i.e., the prepared statement) back to its initial state with one exception: all parameters that had values bound to them retain their values. The statement becomes ready to be reexecuted by the application, and reuses these values in the

reexecution. However, the application may replace these values by new ones executing the `bind` APIs once again before it starts the reexecution.

1.1.1.8. Return values

All API functions return zero or positive integer values. SQLite recommends using mnemonics for return value checks. The return value `SQLITE_OK` indicates a success; `SQLITE_ROW` indicates that the `sqlite3_step` function has found a new row in the rowset that the SELECT statement returned; `SQLITE_DONE` indicates that the statement execution is complete.

The next example presents another SQLite application that can be run from a command line to manipulate databases interactively. It takes two arguments: the first one is a database file name, and the second one an SQL statement. It first opens the database file, then applies the statement to the database by executing the `sqlite3_exec` API function, and finally closes the database file. The function executes the statement directly, without the need to go through the `prepare` and `step` API functions, as was done in the previous example. In case the statement produces output, the `exec` function executes the callback function for each output row. The user must have read permission on the given database file, and depending on the query type, she may need to have a write permission on the file and the directory it's contained in. Here is a command-line based application:

Code View: Scroll / <u>Show All</u>

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

static int callback(void *NotUsed, int argc, char **argv, char **colName)
{
  // Loop over each column in the current row
  int i;
  for (i = 0; i < argc; i++){
    printf("%s = %s\n", colName[i], argv[i] ? argv[i] : "NULL");
  }
  return 0;
}


int main(int argc, char **argv)
{
  sqlite3* db = 0;
  char* zErrMsg = 0;
  int rc;

  if (argc != 3){
    fprintf(stderr, "Usage: %s DATABASE-FILE-NAME SQL-STATEMENT\n", argv[0]);
```

```
        return -1;
    }
    rc = sqlite3_open(argv[1], &db);
    if (rc != SQLITE_OK){
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        return -2;
    }
    rc = sqlite3_exec(db, argv[2], callback, 0, &zErrMsg);
    if (rc != SQLITE_OK){
        fprintf(stderr, "SQL error: %s\n", zErrMsg);
    }
    sqlite3_close(db);
    return rc;
}
```

Compile the preceding application code into an executable, such as `app2`. You can now issue SQL statements that operate on a database. Suppose you are working on the same MyDB database in the current working directory. By executing the following command lines, you can insert new rows in the Students table:

```
./app2 MyDB "insert into Students values(100)"
./app2 MyDB "insert into Students values(10)"
./app2 MyDB "insert into Students values(1000)"
```

If you run the previous application (`app1`) now, you will see the following output:

```
SID = 10
SID = 100
SID = 100
SID = 200
SID = 300
SID = 1000
```

You can also create new tables in databases; for example, `./app2 MyDBExtn "create table Courses(name varchar, SID integer)"` creates the `Courses` table in a new MyDBExtn database in the current working directory.

NOTE

SQLite has an interactive command-line program (*sqlite3*), shown earlier, which you can use to issue SQL commands. You can download a precompiled binary version of it from the SQLite site, or compile it from source. This `app2` example is essentially a bare-bones implementation of *sqlite3*.

SQLite is a thread-safe library. Consequently, many threads in an application can access the same or different databases concurrently.

NOTE

In order to be thread-safe, the SQLite source code must be compiled with the THREADSAFE preprocessor macro set to 1 (enable this by passing `--enable-threadsafe` to the *configure* script when you prepare to compile SQLite). In the default setting, the multithreading feature is turned off for Linux, but turned on for Windows. There is no way you can query the SQLite library to find out whether it has the feature.

This example uses the pthreads library, which is not included with Windows. On Windows, you have two choices: use the tools and compilers from Cygwin (http://www.cygwin.com), or download pthreads for Windows from http://sourceware.org/pthreads-win32/.

Here is a typical multithreaded application:

Code View: Scroll / <u>Show All</u>

```
#include <stdio.h>
#include <pthread.h>
#include "sqlite3.h"

void* myInsert(void* arg)
{
  sqlite3*     db = 0;
  sqlite3_stmt* stmt = 0;
  int val = (int)arg;
  char SQL[100];
  int rc;
  rc = sqlite3_open("MyDB", &db); /* Open a database named MyDB */
  if (rc != SQLITE_OK) {
    fprintf(stderr, "Thread[%d] fails to open the database\n",  val);
    goto errorRet;
  }

  /* Create the SQL string. If you were using string values,
     you would need to use sqlite3_prepare() and sqlite3_bind_*
     to avoid an SQL injection vulnerability. However %d
```

```
   guarantees an integer value, so this use of sprintf is
    safe.
 */
   sprintf(SQL, "insert into Students values(%d)", val);


   /* Prepare the insert statement */
   rc = sqlite3_prepare(db, SQL, -1, &stmt, 0);
   if (rc != SQLITE_OK) {
     fprintf(stderr, "Thread[%d] fails to prepare SQL: %s ->
return code %d\n", val, SQL, rc);
     goto errorRet;
   }
   rc = sqlite3_step(stmt);
   if (rc != SQLITE_DONE) {
     fprintf(stderr,
       "Thread[%d] fails to execute SQL: %s -> return code %d\n", val, SQL, rc);
   }
   else {
     printf("Thread[%d] successfully executes SQL: %s\n", val,
       SQL);
   }
   errorRet:
     sqlite3_close(db);
     return (void*)rc;
}
int main(void)
{
   pthread_t t[10];
   int i;
   for (i=0; i < 10; i++)
     pthread_create(&t[i], 0, myInsert, (void*)i); /* Pass the value of i */

   /* wait for all threads to finish */
   for (i=0; i<10; i++)  pthread_join(t[i], 0);
   return 0;
}
```

This application may not work "out of the box" on Windows and Mac OS X. You may need to recompile SQLite with threading support, and/or obtain the pthread libraries to make the application work on those platforms. Mac OS X includes pthreads, and you can obtain pthreads for

<div style="border:1px solid">

Windows at http://sourceware.org/pthreads-win32/.

</div>

The application creates 10 threads, and each of them tries to insert one row in the Students table in the same MyDB database. SQLite implements a lock-based concurrency scheme, so some or all of the INSERT statements may fail due to lock conflict. The application does not need to worry about concurrency control and database consistency issues; it cannot corrupt the database. SQLite takes care of concurrency control and consistency issues. However, you will need to check for failures, and handle them appropriately within the code (for example, you might retry the command that failed, or inform the user that it failed and let her decide what to do next).

Each thread needs to connect to its database, and needs to create its own SQLite (connection and statement) handles. SQLite does not recommend sharing handles across threads. Even though sharing handles between threads may appear to work, there is no guarantee that you'll get the expected results. In fact, the SQLite library may break and dump core in some versions of Linux. Also, under Unix/Linux systems, you should not attempt to preserve connection handles across a `fork` system call into the child process. Problems such as database corruption or an application crash may arise if you do this.

NOTE

The restriction against sharing a database connection between threads is somewhat relaxed in SQLite 3.3.1 and subsequent versions. Threads can use a database connection safely in mutual exclusion. What this means is that you can switch a connection from one thread to another as long as the former thread does not hold any native file locks on the connection. You can safely assume that no locks are held if the thread has no pending transaction, and if it has reset or finalized all statements on the connection.

Here is a typical SQLite application that uses two databases:

Code View: Scroll / Show All
```c
#include <stdio.h>
#include "sqlite3.h"

int main(void)
{
  sqlite3* db = 0;

  sqlite3_open("MyDB", &db); /* Open a database named MyDB */
  sqlite3_exec(db, "attach database MyDBExtn as DB1", 0, 0, 0);
  sqlite3_exec(db, "begin", 0, 0, 0);
  sqlite3_exec(db, "insert into Students values(2000)", 0, 0, 0);
```

```
    sqlite3_exec(db, "insert into Courses values('SQLite Database', 2000)", 0, 0, 0);
    sqlite3_exec(db, "commit", 0, 0, 0);


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

I have simplified the code by not including error checks. The application opens the MyDB database, and then attaches the MyDBExtn database to the current connection. MyDB needs to have a Students(SID) table, and the MyDBExtn database needs a Courses(name, SID) table. The application opens a transaction by executing the `begin` command, inserts one row in Students and one in Courses inside the transaction, and finally commits the transaction by executing the `commit` command. INSERT statements do not need a callback function, and hence, I pass 0 as the callback parameter in `sqlite3_exec` calls in the example application.

NOTE

SQLite permits multiple SQL statements in a single `exec` API call; the same batch of commands can be executed by passing this sequence of statements in a single `exec` call: `begin; insert into Students values(2000); insert into Courses values('SQLite Database', 2000); commit`. If the batch contains SELECT statements, the same callback function is used to process the resulting rowsets.

A *catalog* is a system table that is created and maintained by SQLite itself. It stores meta information about the database. SQLite maintains one master catalog, named `sqlite_master`, in every database. The catalog stores schema information about tables, indexes, triggers, and views. You can query the master catalog (e.g., `select * from sqlite_master`), but cannot directly modify the catalog. There are other optional catalog tables. All catalog table names start with the `sqlite_` prefix, and the names are reserved by SQLite for internal use. You cannot create database objects (tables, views, indexes, triggers) with such names (in upper, lower or mixed case).

Section 1.2. SQLite Architecture

SQLite consists of seven major component subsystems (also known as modules) partitioned into two divisions: frontend parsing system and backend engine. Two block diagrams, showing the component subsystems and how they interrelate, are given in Figure 1-1.

Figure 1-1. Components of SQLite

(a) Front-end system   (b) Back-end system
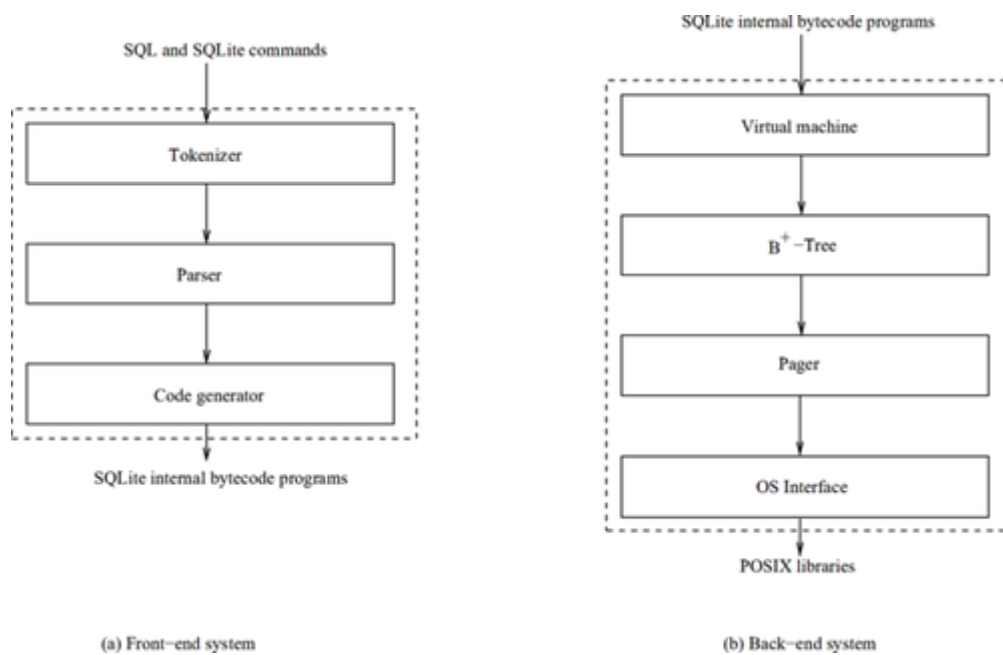
The frontend preprocesses SQL statements and SQLite commands that are sent as input to it by applications. It parses these statements (and commands), optimizes them, and generates equivalent SQLite internal bytecode programs that the backend can execute. The frontend division is composed of three modules:

The tokenizer

The <mark>tokenizer</mark> splits an input statement into tokens.

The parser

The parser analyzes the structure of the statement by analyzing the tokens produced by the tokenizer, and generates a parse tree. The parser also includes an optimizer that restructures the parse tree, and finds an equivalent parse tree that will produce an efficient bytecode program.

The code generator

The code-generator traverses the parse tree, and generates an equivalent bytecode program.

The frontend implements the `sqlite3_prepare` API function.

The backend is the engine that interprets bytecode programs. The engine does the actual database processing work. The backend division is composed of four modules:

The Virtual Machine (VM)

The VM module is the interpreter for the internal bytecode language. It executes bytecode programs to carry out the work of the SQL statements. It is the ultimate manipulator of data from databases. It sees a database as collection of tables and indexes, where a table or index is a set of *tuples*, or *records*.

The B/B$^+$-tree

The B/B$^+$-tree module organizes each tuple set into an ordered tree data structure; tables and indexes in separate B$^+$- and B-trees, respectively. (I will discuss about these data structures in the "Table and Index Management(See 7.)" section.) The module helps the VM to search, insert, and delete tuples in trees. It also helps the VM to create new trees, and to delete old trees.

The pager

The pager module implements a page-oriented database file abstraction on the top of native files. It manages an in-memory cache (of database pages) that the B/B$^+$-tree module uses, and, in addition, manages file locking and journaling to implement transactional ACID properties.

The operating system interface

The operating system interface module provides a uniform interface to different native operating systems.

The backend implements `sqlite3_bind_*`, `sqlite3_step`, `sqlite3_column_*`, `sqlite3_reset` and `sqlite3_finalize API functions`.

NOTE

Because of limited space availability, in this Short Cut I only discuss the SQLite engine, and not the parsing system. The examples in this Short Cut use Linux to present the inner workings of SQLite, but as SQLite is portable to many other operating systems, they should work similarly on whatever platform you use.

Section 1.3. SQLite Limitations

SQLite is different from most other modern SQL databases in that its primary design goal is to be simple. SQLite follows this goal, even if it leads to occasional inefficient implementations of some features. The following is a list of shortcomings of SQLite:

SQL-92 feature

As mentioned previously, SQLite does not support some SQL-92 features that are available in many enterprise database systems. You can obtain the latest information from the SQLite homepage.

Low concurrency

SQLite supports only flat transactions; it does not have nesting and savepoint capabilities. (Nesting means the capability of having subtransactions in a transaction. Savepoints allow a transaction to revert back to previously established states.) It is not capable of ensuring a high degree of transaction concurrency. It permits many concurrent read-transactions, but only one exclusive write-transaction on a single database file. This limitation means that if any transaction is reading from any part of a database file, all other transactions are prevented from writing any part of the file. Similarly, if any one transaction is writing to any part of a database file, all other transactions are prevented from reading or writing any part of the file.

Application restriction

Because of its limited transactional concurrency, SQLite is only good for small-size transactions. In many situations, this is not a problem. Each application does its

database work quickly and moves on, and hence no database is held up by a transaction for more than a few milliseconds. But there are some applications, especially write-intensive ones, that require more transactional concurrency (table or row level instead of database level), and you should use a different DBMS for such applications. SQLite is not intended to be an enterprise DBMS. It is optimal in situations where simplicity of database implementation, maintenance, and administration are more important than the countless complex features that enterprise DBMSs provide.

NFS problems

SQLite uses native file locking primitives to control transaction concurrency. This may cause some problems if database files reside on network partitions. Many NFS implementations are known to contain bugs (on Unix and Windows) in their file locking logic. If file locking does not work the way it is expected, it might be possible for two or more applications to modify the same part of the same database at the same time, resulting in a database corruption. Because this problem arises due to bugs in the underlying filesystem implementation, there is nothing SQLite can do to prevent it.

Another thing is that because of the latency associated with most network filesystems, performance may not be good. In such environments, in cases where the database files must be accessed across the network, a DBMS that implements a client-server model might be more effective than SQLite.

Database size

Because of its developers' engineering design choices, SQLite may not be a good choice for very large databases. In theory, a database file can be as large as two terabytes (2^{41}) long. The logging subsystem has memory overhead that is proportional to the database size. For every write transaction, SQLite maintains one bit of in-memory information for every database page, whether the transaction even reads or writes that page. (The default page size is 1024 bytes.) Thus, memory overhead may become a severe bottleneck for databases that have more than a few million pages.

Number and type of objects

A table or index is limited by at most $2^{64} - 1$ entries. (Of course, you cannot have this many entries because of the 2^{41} bytes database size limit.) A single entry can

hold up to 2^{30} bytes of data in SQLite's current implementation. (The underlying file format supports row sizes up to about 2^{62} bytes of data.) Upon opening a database file, SQLite reads and preprocesses all entries from the master catalog table and creates many in-memory catalog objects. So, for best performance, it is better to keep down the number of tables, indexes, views, and triggers. Likewise, though there is no limit on the number of columns in a table, more than a few hundred seems extreme. Only the first 31 columns of a table are candidates for certain optimizations. You can put as many columns in an index as you like, but indexes with more than 30 columns will not be used to optimize queries.

Host variable reference

In some embedded DBMSs, SQL statements can reference host variables (i.e., those from application space) directly. This is not possible in SQLite. SQLite instead permits binding of host variables to SQL statements using `sqlite3_bind_*` API functions for input parameters, and not for output values. This approach is generally better than the direct access approach because the latter requires a special preprocessor that converts SQL statements into special API calls.

Stored procedure

Many DBMSs have capability to create and store what are called stored procedures. A *stored procedure* is a group of SQL statements that form a logical unit of work and perform a particular task. SQL queries can use these procedures. SQLite does not have this capability.

Chapter 2. Database File Format

Before I go into details of the SQLite engine, I will first present database naming conventions and database file structure in the following two subsections.

2.1. Database Naming Conventions

When an application tries to open a database by making a call to the `sqlite3_open` API function, it does so by giving the function the name of the database file. The file name can be the relative path name with respect to the current working directory, or the full path name starting from the root of the system file tree. Any regular file name accepted by the native file system is good. There are, however, two notable exceptions:

- If the file name is a C language NULL pointer (i.e., 0), SQLite creates and opens a *new* temporary file;
- If the file name is "**:memory**:" SQLite creates an in-memory database.

In either exception case, when the application closes the database connection, the database becomes extinct, i.e., the database is not persistent.

NOTE

SQLite chooses all temporary file names at random. The names start with `sqlite_` followed by 16 random alphanumeric characters. The files are stored in a standard native temporary file directory. SQLite tries out directories in the order (1) */var/tmp*, (2) */usr/tmp*, (3) */tmp*, (4) the current working directory.

In either of the above-mentioned ways of opening databases (file, temporary database, or in-memory database), the database (created and/or) opened by SQLite is internally named as the `main` database.

NOTE

Internally, the database file name is not a database name. They are two different but related concepts in SQLite. By using the SQLite `attach` command, you can associate the same database file to a database connection as different database names. You can apply operations on the same file via these database names. You may refer to the SQLite homepage to know more about the semantics of `attach`.

SQLite maintains a separate temporary database for each database connection the application has opened with `sqlite3_open`; the database is named the `temp` database. The temp database stores temporary objects such as tables and their indexes. (Applications can use these two names, i.e., `main` and `temp`, in their queries. For example, `select * from temp.table1` returns all rows from `table1` in the `temp` database and not from the `main` database. The catalog name for the `temp` database is `sqlite_temp_master`.) The temporary objects are only visible within the same database connection (not in other connections to the same database file in the same thread, process, or other processes). SQLite stores the `temp` database in a separate temporary file that is distinct from the `main` database file. It deletes the temporary file when the application closes the connection to the `main` database.

Section 2.1. Database Naming Conventions

Chapter 2. Database File Format

Before I go into details of the SQLite engine, I will first present database naming **conventions** and database file structure in the following two subsections.

2.1. Database Naming Conventions

When an application tries to open a database by making a call to the `sqlite3_open` API function, it does so by giving the function the name of the database file. The file name can be the relative path name with respect to the current working directory, or the full path name starting from the root of the system file tree. Any regular file name accepted by the native file system is good. There are, however, two notable exceptions:

- If the file name is a C language NULL pointer (i.e., 0), SQLite creates and opens a *new* temporary file;
- If the file name is ":memory:" SQLite creates an in-memory database.

In either exception case, when the application closes the database connection, the database becomes extinct, i.e., the database is not persistent.

NOTE

SQLite chooses all temporary file names at random. The names start with `sqlite_` followed by 16 random alphanumeric characters. The files are stored in a standard native temporary file directory. SQLite tries out directories in the order (1) *var/tmp*, (2) */usr/tmp*, (3) */tmp*, (4) the current working directory.

In either of the above-mentioned ways of opening databases (file, temporary database, or in-memory database), the database (created and/or) opened by SQLite is internally named as the `main` database.

NOTE

Internally, the database file name is not a database name. They are two different but related concepts in SQLite. By using the SQLite `attach` command, you can associate the same database file to a database connection as different database names. You can apply operations on the same file via these database names. You may refer to the SQLite homepage to know more about the semantics of `attach`.

SQLite maintains a separate temporary database for each database connection the application has opened with `sqlite3_open`; the database is named the `temp` database. The temp database stores temporary objects such as tables and their indexes. (Applications can use these two names, i.e., `main` and `temp`, in their queries. For example, `select * from temp.table1` returns all rows from `table1` in the `temp` database and not from the `main` database. The catalog name for the `temp` database is `sqlite_temp_master`.) The temporary objects are only visible within the same database connection (not in other connections to the same database file in the same thread, process, or other processes). SQLite stores the `temp` database in a separate temporary file that is distinct from the `main` database file. It deletes the temporary file when the application closes the connection to the `main` database.

Section 2.2. Database File Structure

Except in-memory databases, SQLite stores an entire (main or temp) database in a single database file.

For ease in space management and reading/writing of data from databases, SQLite divides each database (including an in-memory database) into fixed-size regions called *database pages*, or simply *pages*. The page size is a power of 2, and can be between 512 and 32,768 (both inclusive); the default value is 1,024. (The upper bound is a limit imposed by the necessity of storing page size in 2-byte signed integer variables in various places in code and external storage.) The database is an (expandable and contractible) array of pages. An index to the page array is called a *page number*. Page numbers start at 1, and can go sequentially up to 2,147,483,647 ($2^{31}-1$). (The upper bound may be restricted further due to the maximum file size limit enforced by the native filesystem.) Page number 0 is treated as the NULL page or "not a page"—the page does not exist physically. Page 1 and onward are stored one after another into the database file starting at file offset 0.

NOTE

Once a database file is created, SQLite uses the compile time default page size, but the size can be changed by a pragma command before creating the first table in the database. SQLite stores the size value as a part of metadata. It will use this page size value instead of the default one. (As mentioned previously, pragma commands are used to modify the behavior of the SQLite library. See the SQLite homepage for details.)

There are four types of pages: leaf, internal, overflow, and free. *Free pages* are inactive (currently unused) pages; the others are active pages. B^+-*tree internal pages* contain navigational information for searches (B-tree internal pages have search information and data). *Leaf pages* store actual data (e.g., table rows) in B^+-trees. If a row's data is too large to fit in a single page, part of the data is stored in the tree page, and the remaining part in *overflow pages*.

SQLite can use any database page for any page type, except Page 1, which is always a B^+-tree internal page. The page also contains a 100 byte file header record that is stored starting at file offset 0. The header information characterizes the structure of the database file. SQLite initializes the header when it creates the file. The format of the file header is given in the following table. The first two columns in Table 2-1 are in bytes.

Table 2-1. Structure of database file header

Offset	Size	Description
0	16	Header string
16	2	Page size in bytes
18	1	File format write version
19	1	File format read version
20	1	Bytes reserved at the end of each page
21	1	Max embedded payload fraction
22	1	Min embedded payload fraction
23	1	Min leaf payload fraction
24	4	File change counter
28	4	Reserved for future use
32	4	First freelist page
36	4	Number of freelist pages
40	60	15 4-byte meta values

Here is a description of each header element:

Header string

This is the 16 byte string: "SQLite format 3."

Page size

This is the size of each page in this database.

File format

The two bytes at offsets 18 and 19 are used to indicate the file format version. They both have to be 1 in the current version of SQLite, or an error is returned. If

future file format changes occur, these numbers will increase to indicate the new file format version number.

Reserved space

SQLite may reserve a small fixed amount of space (<= 255 bytes) at the end of each page for its own purpose, and this value is stored at offset 20; the default value is 0. It is nonzero when a database uses SQLite's built-in encryption technology. The first part of a page (page size minus reserved size) is the usable space where database content proper is stored.

Embedded payload

The *max embedded payload fraction* value (at offset 21) is the amount of the total usable space in a page that can be consumed by a single entry (called a cell or record) of a standard B/B$^+$-tree internal node. A value of 255 means 100 percent. The default max embedded payload fraction value is 64 (i.e., 25 percent): the value is to limit the maximum cell size so that at least 4 cells fit on one node. If the payload for a cell is larger than the max value, then extra payload is spilled into overflow pages. Once SQLite allocates an overflow page, it moves as many bytes as possible into the overflow page without letting the cell size to drop below the *min embedded payload fraction* value (at offset 22). The default value is 32, i.e., 12.5 percent.

The *min leaf payload fraction* value (at offset 23) is like the min embedded payload fraction, except that it is for B$^+$-tree leaf pages. The default value is 32, i.e., 12.5 percent. The max payload fraction value for a leaf node is always 100 percent (or 255), and is not specified in the header. (There are no special-purpose leaf nodes in B-trees.)

File change counter

The *file change counter* (at offset 24) is used by transactions. That value is incremented by every transaction. This value is intended to indicate when the database has changed so that the pager can avoid having to flush its cache, though that feature has not been implemented as of this writing. The pager is responsible for incrementing this value.

The freelist of unused pages originates in the file header at offset 32. The total number of free pages is stored at offset 36. The freelist is organized in a rooted trunk (see Figure 2-1). Freelist pages come in two subtypes: trunk pages and leaf pages. The file header points to the first one on the linked list of trunk pages. Each trunk page points to multiple leaf pages. (A leaf page content is unspecified.)

Figure 2-1. Structure of a freelist
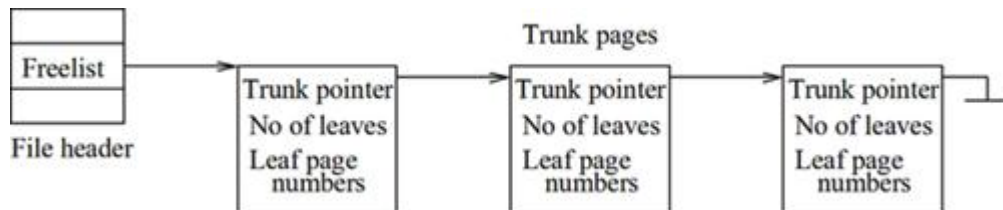
A trunk page is formatted like the following, starting at the base of the page:

- A 4-byte page number of the next trunk page
- A 4-byte integer value to indicate the number of leaf pointers stored on this page
- Zero or more 4-byte page numbers for leaf pages

When a page becomes inactive, SQLite adds it to the freelist, and does not release it to the native filesystem. When you add new information to the database, SQLite takes out free pages off the freelist to store the information. If the freelist is empty, SQLite acquires new pages from the native filesystem, and appends them to the database file.

NOTE

You can purge the freelist by executing the `vacuum` command on the database. The command makes a copy of the database into a temporary file (the copy is made using `INSERT INTO ...SELECT * FROM...` commands). Then, it overwrites the original database with the temporary copy, under the protection of a transaction.

Meta variables

At offset 40, there are fifteen 4-byte integer values that are reserved for the B^+-tree and the VM (virtual machine) modules. They represent values of many meta variables, including the database schema cookie number at offset 40; this value is incremented at each schema change. Other meta variables include the file formatting information of the schema layer at offset 44, the page cache size at 48, the autovacuum flag at 52, the text encoding (1:UTF-8, 3:UTF-16 LE, 4:UTF-16 BE) at 56, and the user version number at 60. You can find more information about these variables in the SQLite source files, notably _btree.c_.

NOTE

The database file format for SQLite is backward compatible back to version 3.0.0. This means that any version of SQLite can read and write a database file that was originally created by version 3.0.0. This is mostly true in the other direction—version 3.0.0 of SQLite can usually read and write any SQLite database created by later versions of the library. However, there are some new features introduced by later versions of SQLite that version 3.0.0 does not understand, and if the database contains these optional new features, older versions of the library will not be able to read and understand it.

The file header is followed by a B$^+$-tree internal node on Page 1. The node is the root of the master catalog table, named `sqlite_master` or `sqlite_temp_master` for a regular (main or attached) or temp database, respectively.

NOTE

All multibyte integer values are stored in the big-endian (most significant byte first) order. This lets you safely move your database files from one platform to another.

Chapter 3. Page Cache Management

The pager is the only module that accesses (through your operating system's native IO APIs) native database and journal files. But, it neither interprets the content of databases, nor modifies the content on its own. (The pager may modify some information in the file header record, such as the file change counter.) It takes the usual random access/byte-oriented filesystem operations, and abstracts them into a random access page-based system for working with database files. It defines an easy-to-use, filesystem-independent interface for accessing pages from database files. The B$^+$-tree module always uses the pager interface to access databases, and never directly accesses any database or journal file. It sees the database file as a logical array of (uniform size) pages.

Databases (except in-memory ones) normally reside on external storage devices like a disk, in the form of ordinary native files. When SQLite needs a data item, it reads it from the database file into the main memory, manipulates it in-memory, and, if needed, writes it back to the file. Normally, databases are very large compared to the available main memory. Because of the limited availability of the main memory, only a part of the memory is reserved to hold a (tiny) fraction of data from database file(s), and this reserved memory space is popularly called a database cache or data buffer; in SQLite terminology, it's called the *page cache*. The pager is the cache manager, among many other things (described next).
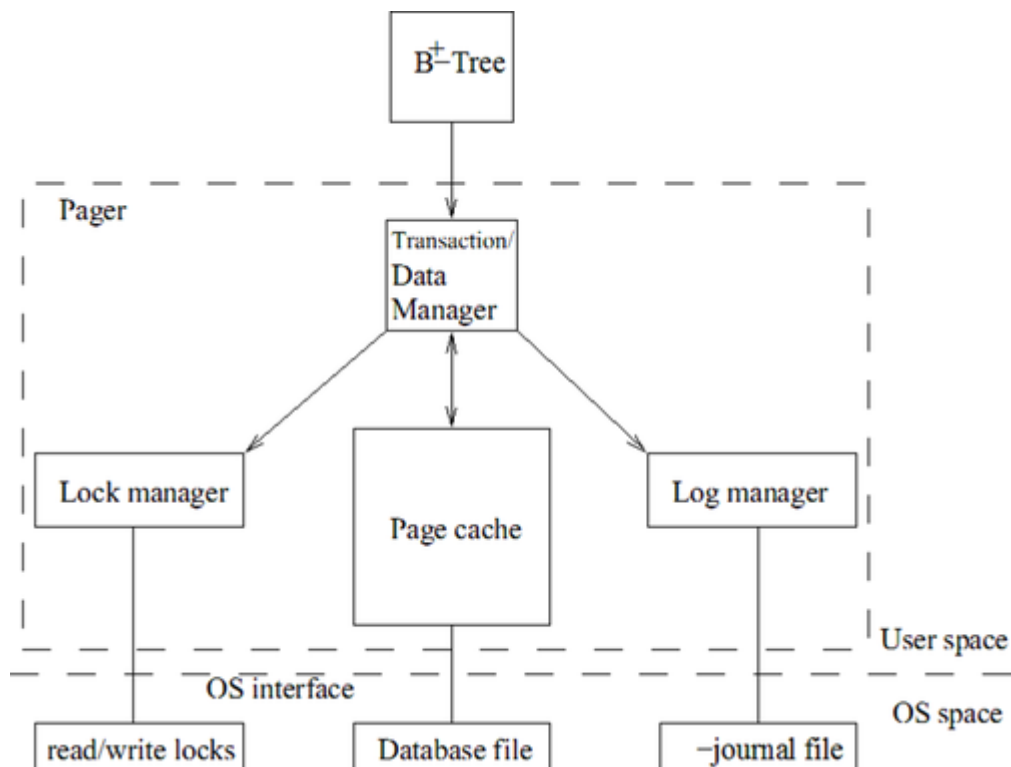
3.1. Pager Responsibilities

For each database file, moving pages between the file and the cache is the basic function of the pager as the cache manager. The page movement is transparent to the B$^+$-tree and

higher-level modules. The pager is the mediator between the native filesystem and those modules. Its main purpose is to make database pages addressable in the main memory so that those modules can access the page contents directly. It also coordinates the writing of pages back to the database file. It creates an abstraction so that the entire database file appears to reside in the main memory as an array of pages.

Apart from the cache management work, the pager does carry out many other functions of a typical DBMS. It provides the core services of a typical transaction processing system: transaction management, data management, log management, and lock management. As a transaction manager, it implements transactional ACID properties by taking charge of concurrency control and recovery. It is responsible for atomic commit and rollback of transactions. As a data manager, it coordinates reading and writing of pages in database files with the cache and does file space management work. As a log manager, it decides on the writing of log records in journal files. As a lock manager, it makes sure that transactions, before accessing a database page, have appropriate locks on the database file. In a nutshell, the pager module implements storage persistence and transactional atomicity. The interconnections between all the submodules of the pager are shown in Figure 3-1.

Figure 3-1. Interconnection of pager submodules

NOTE

All modules above the pager are completely insulated from low-level lock and log management mechanisms. In fact, they are not aware of locking and logging activities. The B$^+$-tree module sees every thing in terms of transactions, and is not concerned with how the transactional ACID

properties are implemented. The pager module splits the activities of a transaction into locking, logging, and reading and writing of database files. The B$^+$-tree module requests a page from the pager by the page number. The pager, in turn, returns a pointer to the page data loaded into the page cache. Before modifying a page, the B$^+$-tree module informs the pager so that it (the pager) can save sufficient information (in a journal file) for possible use in future recovery, and can acquire appropriate locks on the database file. The B$^+$-tree module eventually notifies the pager when it (the B$^+$-tree) has finished using a page; the pager handles writing the page back to the file if the page was modified.

In this section, I present page cache management, and in the next, transaction management.

Section 3.1. Pager Responsibilities

Chapter 3. Page Cache Management

The pager is the only module that accesses (through your operating system's native IO APIs) native database and journal files. But, it neither interprets the content of databases, nor modifies the content on its own. (The pager may modify some information in the file header record, such as the file change counter.) It takes the usual random access/byte-oriented filesystem operations, and abstracts them into a random access page-based system for working with database files. It defines an easy-to-use, filesystem-independent interface for accessing pages from database files. The B$^+$-tree module always uses the pager interface to access databases, and never directly accesses any database or journal file. It sees the database file as a logical array of (uniform size) pages.

Databases (except in-memory ones) normally reside on external storage devices like a disk, in the form of ordinary native files. When SQLite needs a data item, it reads it from the database file into the main memory, manipulates it in-memory, and, if needed, writes it back to the file. Normally, databases are very large compared to the available main memory. Because of the limited availability of the main memory, only a part of the memory is reserved to hold a (tiny) fraction of data from database file(s), and this reserved memory space is popularly called a database cache or data buffer; in SQLite terminology, it's called the *page cache*. The pager is the cache manager, among many other things (described next).

3.1. Pager Responsibilities

For each database file, moving pages between the file and the cache is the basic function of the pager as the cache manager. The page movement is transparent to the B$^+$-tree and higher-level modules. The pager is the mediator between the native filesystem and those modules. Its main purpose is to make database pages addressable in the main memory so that those modules can access the page contents directly. It also coordinates the writing of pages back to the database file. It creates an abstraction so that the entire database file appears to reside in the main memory as an array of pages.

Apart from the cache management work, the pager does carry out many other functions of a typical DBMS. It provides the core services of a typical transaction processing system: transaction management, data management, log management, and lock management. As a transaction manager, it implements transactional ACID properties by taking charge of concurrency control and recovery. It is responsible for atomic commit and rollback of transactions. As a data manager, it coordinates reading and writing of pages in database files with the cache and does file space management work. As a log manager, it decides on the writing of log records in journal files. As a lock manager, it makes sure that transactions, before accessing a database page, have appropriate locks on the database file. In a nutshell, the pager module implements storage persistence and transactional atomicity. The interconnections between all the submodules of the pager are shown in Figure 3-1.

Figure 3-1. Interconnection of pager submodules

NOTE

All modules above the pager are completely insulated from low-level lock and log management mechanisms. In fact, they are not aware of locking and logging activities. The B^+-tree module sees every thing in terms of transactions, and is not concerned with how the transactional ACID properties are implemented. The pager module splits the activities of a transaction into locking, logging, and reading and writing of database files. The B^+-tree module requests a page from the pager by the page number. The pager, in turn, returns a pointer to the page data loaded into the page cache. Before modifying a page, the B^+-tree module informs the pager so that it (the pager) can save sufficient information (in a journal file) for possible use in future recovery, and can acquire appropriate locks on the database file. The B^+-tree module eventually notifies

the pager when it (the B$^+$-tree) has finished using a page; the pager handles writing the page back to the file if the page was modified.

In this section, I present page cache management, and in the next, transaction management.

Section 3.2. Pager Interface Structure

The pager module implements a data structure named `Pager`. Each open database file is managed through a separate `Pager` object, and each `Pager` object is associated with one and only one instance of an open database file. (The object is synonymous with the database file.) The B$^+$-tree module, to use a database file, creates a new `Pager` object, and uses the object as a handle to apply pager-level operations on the file. The pager module uses the handle to track down information regarding file locks, the journal file, the status of the database, the status of the journal, etc.

Section 3.3. Cache Management

SQLite maintains a separate page cache for each open database file. If a thread opens the same file two or many times, the pager creates and initializes a separate page cache for the file only at the first open call. If two or more threads open the same file, there will be many independent caches for the same file. In-memory databases do not refer to any external storage devices. But, they are also treated like ordinary native files, and are stored entirely within the cache. Thus, the B/B$^+$-tree module uses the same interface to access either type of database.

NOTE

Page caches reside in an application's memory space. Note that the same pages may be cached by the native operating system. When an application reads a piece of data from any file, the operating system normally makes its own copy of the data first, and then a copy in the application. We are not interested in knowing how the operating system manages its own cache. SQLite's page cache organization and management are independent of those of the native operating system.

Management of page cache is crucial for system performance. The following subsections discuss how the pager organizes and maintains page cache, and how cache clients read and modify cache elements.

In general, to speed up searching a cache, the currently held in-cache items are well organized. SQLite uses a hash table to organize cached pages, and uses *page-slots* to hold pages in the table. The cache is fully associative, that is, any slot can store any page. The hash table is initially empty. As page demand increases, the pager creates new slots and inserts them in the hash table. There is a maximum limit on the number of slots a cache can have. (In-memory

databases have no such limit so long as the native operating system allows the application space to grow.)
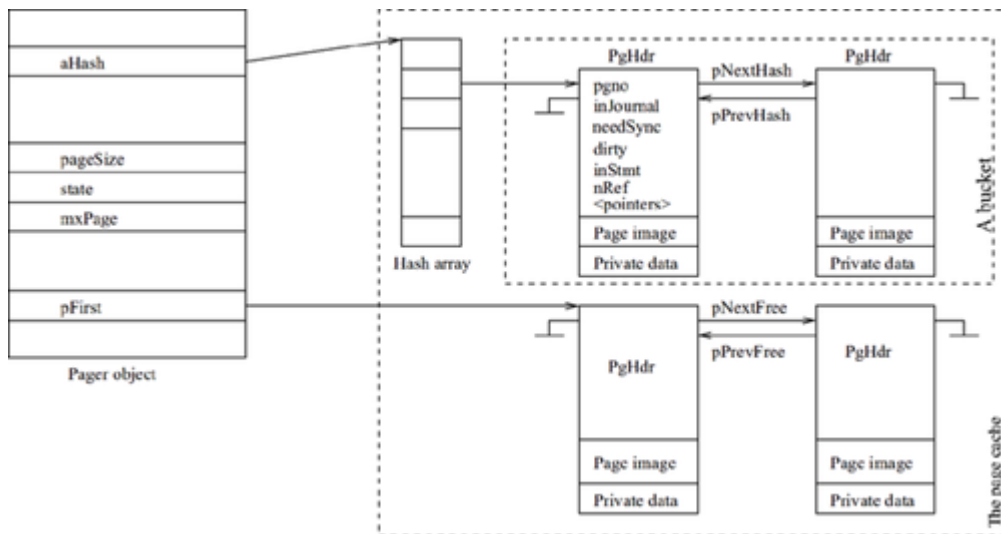
Figure 3-2. Page cache

Figure 3-2 depicts a typical cache layout. Each page in the hash table is represented by a header object of `PgHdr` type. The page image is stored after the `PgHdr` object. The image is followed by a piece of private data that is used by the B$^+$-tree module to keep page-specific control information there. (In-memory databases have no journal file, so their recovery information is recorded in in-memory objects. Pointers to those objects are stored following the private part: these pointers are used by the pager only.) This (additional nonpage) space is initialized to zeros when the pager brings the page into the cache. All pages in the cache are accessible via a hash array, namely `aHash` in the `Pager` object; the array size is fixed when the SQLite library is compiled. Each array element points to a "bucket" of pages; pages in each bucket are organized in an unordered doubly linked list.

The `PgHdr` is only visible to the pager module, and not visible to the B$^+$-tree and higher-level modules. The header has many control variables. The `pgno` variable identifies the page number of the database page it represents. The `injournal` variable is true if the page has already been written to the rollback journal. The `needSync` variable is true if the journal needs a flush before writing this page back in the database file. (A *flush* on a file transfers modified parts of the file to the disk surface.) The `dirty` variable is true if the page has been modified, and the new value is not yet written back to the database file. The `inStmt` variable is true if the page is in the current statement journal. The `nRef` variable is the reference count on this page. If the `nRef` value is greater than zero, the page is in active use, and we say that the page is *pinned down*; otherwise, the page is unpinned and free. There are many pointer variables in `PgHdr` object (not all of them are shown in the figure). The `pNextHash` and `pPrevHash` pointers are used to link together pages in the same hash bucket. The `pNextStmt` and `pPrevStmt` pointers link together those pages that are in the statement journal. (I talk about the statement journal later.) The `pNextFree` and `pPrevFree` pointers are used to link together all free pages. Free pages are never taken out of hash buckets, even though the above figure seems to suggest

otherwise. All pages (free or not) in the cache are linked together through the `pNextAll` pointer. The `pDirty` pointer links together all dirty pages. Note that a free page can be dirty, too.

The cache is referenced by using a search key—the page number. To read a page, the client B$^+$-tree module invokes the `sqlite3pager_get` pager API function on the page number. The function performs the following steps for a page *P:*

1. It searches the cache space.
    a. It applies a hash function on P and gets an index. (SQLite uses a very simple hash function to determine the index value: page number modulo the size of the aHash array.)
    b. It uses the index into the aHash array and gets the hash bucket.
    c. It searches the bucket by chasing the pNextHash pointers. If P is found there, we say a cache hit has occurred. It pins down (i.e., increments the nRef value by 1) the page and returns the address of the page to the caller.
2. If *P* is not found in the cache, it is considered a cache miss. The function looks for a free slot that can be used to load the desired page. (If the cache has not reached the maximum limit, it instead creates a new free slot.)
3. If no free slot is available or can be created, it determines a slot from which the current page can be released to reuse the slot. This is called a *victim* slot.
4. If the victim (or the free slot) is dirty, it writes the page to the database file. (Following write-ahead-log (WAL) protocol, it flushes the journal file.)
5. It reads page *P* from the database file into the free slot, pins down the page (i.e., it sets the `nRef` value to 1), and returns the address of the page to the caller. If *P* is greater than the current max page in the file, it does not read the page; instead, it initializes the page to zeros. It also initializes the bottom private part to zeros whether it reads the page from the file.

SQLite strictly follows fetch-on-demand policy to keep the page fetch logic very simple.

When a page address is returned to the client B$^+$-tree module, the pager does not know when the client actually works on the page. SQLite follows this standard protocol for each page: the client acquires the page, uses the page, and then releases the page. After acquiring a page, the client can directly modify the content of the page, but it must call the `sqlite3pager_write` pager API function on the page prior to making any modifications there. On return from the call, the client can update the page in place.

The first time the `write` function is called on a page, the pager writes the original content of the page into the rollback journal file as part of a new log record, and sets the `injournal` and `needSync` flags on. Later, when the log record is flushed onto the disk surface, the pager clears

the `needSync` flag. (SQLite follows WAL protocol: it does not write a modified page back into the database file until the corresponding `needSync` has been cleared.) Every time the `write` function is called on a page, the `dirty` flag is set; the flag is cleared only when the pager writes back the page content to the database file. Because the time when the client modifies a page is not known to the pager, updates on the page are not immediately propagated to the database file. The pager follows a delayed write (write-back) page update policy. The updates are propagated to the database file only when the pager performs a cache flush or selectively recycles dirty pages.

Cache replacement refers to the activity that takes place when a cache becomes full, and old pages are removed from the cache to make room for new ones. As mentioned in the "Cache read" section, when a requested page is not in the cache and a free slot is not available in the cache, the pager victimizes a slot for replacement. You may recall that the page cache is fully associative, that is, any slot is good for the new page. The slot that is victimized is dictated by the cache replacement policy. SQLite follows a kind of least recently used (LRU) cache replacement policy.

SQLite organizes free pages in a logical queue. When a page is unpinned, the pager appends the page at the tail end of the queue. The victim is chosen from the header end of the queue, but may not always be the head element on the queue as is done in the pure LRU. SQLite tries to find a slot on the queue starting at the head such that recycling that slot would not involve doing a flush of the journal file. (Following the WAL protocol, before writing a dirty page to the database file, the pager flushes the journal file. Flushing is a slow operation, and SQLite tries to postpone the operation as long as possible.) If such a victim is found, the foremost one on the queue is recycled. Otherwise, SQLite flushes the journal file first, and then recycles the head slot from the queue. If the victim page is dirty, the pager writes the page to the database file before recycling it.

NOTE

Pinned pages are currently in active use and cannot be recycled. To avoid the scenario in which all pages in the cache are pinned down, SQLite needs a minimum number of slots in the cache so that it always has some cache slot(s) to recycle; the minimum value is 10 as of the SQLite 3.3.6 release.

Chapter 4. Transaction Management

Managing transactions is crucial for database consistency. Database systems implement ACID properties to ensure consistencies. SQLite relies on native file locking and page journaling to implement ACID properties. You may recall that SQLite supports only flat transactions; it does not have nesting and savepoint capabilities.

4.1. Transaction Types

SQLite executes each SQL statement in a transaction. It supports both read- and write-transactions. Applications cannot read data from a database except in a read- or write-transaction, and they cannot write into a database except in a write-transaction. They do not need to explicitly tell SQLite to execute individual SQL statements within transactions. SQLite automatically does so; this is the default behavior, and the system is said to be in *autocommit* mode. Those transactions are called automatic or system level transactions. For a SELECT statement, SQLite creates a read-transaction. For a non-SELECT statement, SQLite first creates a read-transaction, and then converts it into a write-transaction. Each transaction is automatically committed (or aborted) at the end of the statement execution. Applications are not aware of system transactions. They submit SQL statements to SQLite, which takes care of the rest as far as ACID properties are concerned. Applications receive back outcome of the SQL execution from SQLite. An application can initiate concurrent executions of SELECT statements (read-transactions) on the same database connection, but it can initiate one non-SELECT (write-transaction) on an idle connection.

The autocommit mode might be expensive for some applications, especially for those that are highly write intensive, because SQLite requires reopening, writing to, and closing the journal file for each non-SELECT statement. You will see shortly that SQLite discards the page-cache at the end of each statement (including SELECT) execution in the autocommit mode. It rebuilds the cache for each statement execution. Cache rebuilding is a costly, inefficient action, as it involves doing I/O from disk. In addition, there is also concurrency control overhead, as applications need to reacquire and release locks on database files for each SQL statement execution. This overhead can incur a significant performance penalty (especially for large applications), and can only be curtailed by opening a user level transaction surrounding many SQL statements, as shown in the "Working with multiple databases(See 3.1)" section. Here's another simple SQLite application that creates a user transaction:

```
BEGIN;
    INSERT INTO tablel values(100);
    INSERT INTO table2 values(20, 100);
    UPDATE tablel SET x=x+1 WHERE y> 10;
    INSERT INTO table3 VALUES (1,2,3);
COMMIT;
```

An application can manually start a new transaction by explicitly executing a BEGIN command. The transaction is referred to as a *user level transaction* (or simply a user transaction). When this is done, SQLite leaves the default autocommit mode; it does not invoke a commit or abort at the end of each SQL statement execution, nor does it discard the page cache. Successive SQL statements become a part of the user transaction. SQLite commits (respectively, aborts) the transaction when the application executes the COMMIT (respectively, ROLLBACK) command. If the transaction aborts or fails, or the application closes the connection, the entire

transaction is rolled back. SQLite reverts back to the autocommit mode on completion of the transaction.

NOTE

SQLite supports only flat transactions. Also, an application cannot open more than one user transaction on a database connection at a time. You'll get an error if you execute a BEGIN command inside a user transaction.

A user transaction is a little more than a flat transaction. Each non-SELECT statement in the transaction is executed in a separate statement level subtransaction. Although SQLite does not have the general savepoint capability, it does, however, ensure savepoint for the current statement subtransaction. If the current statement execution fails, SQLite does not abort the user transaction. It instead restores the database to the state prior to the start of the statement execution; the transaction continues from there. The failed statement does not alter the outcome of other previously executed SQL statements, or the new ones, unless the main user transaction aborts itself. SQLite helps a long user transaction to withstand some statement failures amicably.

In the above example, each of the four SQL statements is executed in a separate subtransaction, one after another. If, for example, a constraint error occurs on the tenth row of the UPDATE, all previous nine rows of that update will be rolled back, but the changes from the three INSERTs (surrounding the UPDATE) will be committed when the application executes the COMMIT command.

Section 4.1. Transaction Types

Chapter 4. Transaction Management

Managing transactions is crucial for database consistency. Database systems implement ACID properties to ensure consistencies. SQLite relies on native file locking and page journaling to implement ACID properties. You may recall that SQLite supports only flat transactions; it does not have nesting and savepoint capabilities.

4.1. Transaction Types

SQLite executes each SQL statement in a transaction. It supports both read- and write-transactions. Applications cannot read data from a database except in a read- or write-transaction, and they cannot write into a database except in a write-transaction. They do not need to explicitly tell SQLite to execute individual SQL statements within transactions. SQLite automatically does so; this is the default behavior, and the system is said to be in *autocommit* mode. Those transactions are called automatic or system level transactions. For a SELECT statement, SQLite creates a read-transaction. For a non-SELECT statement, SQLite first creates a read-transaction, and then converts it into a write-transaction. Each transaction is automatically committed (or aborted) at the end of the statement execution. Applications are

not aware of system transactions. They submit SQL statements to SQLite, which takes care of the rest as far as ACID properties are concerned. Applications receive back outcome of the SQL execution from SQLite. An application can initiate concurrent executions of SELECT statements (read-transactions) on the same database connection, but it can initiate one non-SELECT (write-transaction) on an idle connection.

The autocommit mode might be expensive for some applications, especially for those that are highly write intensive, because SQLite requires reopening, writing to, and closing the journal file for each non-SELECT statement. You will see shortly that SQLite discards the page-cache at the end of each statement (including SELECT) execution in the autocommit mode. It rebuilds the cache for each statement execution. Cache rebuilding is a costly, inefficient action, as it involves doing I/O from disk. In addition, there is also concurrency control overhead, as applications need to reacquire and release locks on database files for each SQL statement execution. This overhead can incur a significant performance penalty (especially for large applications), and can only be curtailed by opening a user level transaction surrounding many SQL statements, as shown in the "Working with multiple databases(See 3.1)" section. Here's another simple SQLite application that creates a user transaction:

```
BEGIN;
    INSERT INTO table1 values(100);
    INSERT INTO table2 values(20, 100);
    UPDATE table1 SET x=x+1 WHERE y> 10;
    INSERT INTO table3 VALUES (1,2,3);
COMMIT;
```

An application can manually start a new transaction by explicitly executing a BEGIN command. The transaction is referred to as a *user level transaction* (or simply a user transaction). When this is done, SQLite leaves the default autocommit mode; it does not invoke a commit or abort at the end of each SQL statement execution, nor does it discard the page cache. Successive SQL statements become a part of the user transaction. SQLite commits (respectively, aborts) the transaction when the application executes the COMMIT (respectively, ROLLBACK) command. If the transaction aborts or fails, or the application closes the connection, the entire transaction is rolled back. SQLite reverts back to the autocommit mode on completion of the transaction.

NOTE

SQLite supports only flat transactions. Also, an application cannot open more than one user transaction on a database connection at a time. You'll get an error if you execute a BEGIN command inside a user transaction.

A user transaction is a little more than a flat transaction. Each non-SELECT statement in the transaction is executed in a separate statement level subtransaction. Although SQLite does not have the general savepoint capability, it does, however, ensure savepoint for the current

statement subtransaction. If the current statement execution fails, SQLite does not abort the user transaction. It instead restores the database to the state prior to the start of the statement execution; the transaction continues from there. The failed statement does not alter the outcome of other previously executed SQL statements, or the new ones, unless the main user transaction aborts itself. SQLite helps a long user transaction to withstand some statement failures amicably.

In the above example, each of the four SQL statements is executed in a separate subtransaction, one after another. If, for example, a constraint error occurs on the tenth row of the UPDATE, all previous nine rows of that update will be rolled back, but the changes from the three INSERTs (surrounding the UPDATE) will be committed when the application executes the COMMIT command.

Section 4.2. Lock Management

SQLite, to produce serializable execution of transactions, uses a locking mechanism to regulate database access requests from transactions. It follows strict two-phase locking protocol, i.e., it releases locks only on transaction completion. SQLite does database level locking, but neither row nor page nor table level locking: it sets locks on the entire database file, and not on particular data items in the file.

NOTE

A subtransaction acquires locks through the container user transaction. All locks are held by the user transaction until it commits or aborts irrespective of the outcome of the subtransaction.

From the viewpoint of a single transaction, a database file can be in one of the following five locking states:

NOLOCK

> The transaction does not hold a lock on the database file. It can neither read nor write the database file. Other transactions can read or write the database as their own locking states permit. Nolock is the default state when a transaction is initiated on a database file.

SHARED

This lock only permits reading from the database file. Any number of transactions can hold shared locks on the file at the same time, and hence there can be many simultaneous read-transactions. No transaction is allowed to write the database file while one or more transactions hold shared locks on the file.

RESERVED

This lock permits reading from the database file. A reserved lock means that the transaction is planning on writing the database file at some point in the future, but that it is currently just reading the file. There can be at most one reserved lock on the file, but the lock can coexist with any number of shared locks. Other transactions may obtain new shared locks on the file, but not other locks.

PENDING

This lock permits reading from the database file. A pending lock means that the transaction wants to write to the database as soon as possible. It is just waiting on all current shared locks to clear prior to its acquiring an exclusive lock. There can be at most one pending lock on the file, but the lock can coexist with any number of shared locks with one difference: other transactions may hold on to existing shared locks, but may not obtain new (shared or other) ones.

EXCLUSIVE

This is the only lock that permits writing (and reading too) the database file. Only one exclusive lock is allowed on the file, and no other lock of any kind is allowed to coexist with an exclusive lock.

The lock compatibility matrix is given in Table 4-1. The rows are the existing lock mode on the database held by a transaction, and the columns are a new request from another transaction. Each matrix cell identifies the compatibility between the two locks: Y indicates the new request can be granted; N indicates that it cannot.

Table 4-1. Lock compatibility matrix

Requested lock ➝	SHARED	RESERVED	PENDING	EXCLUSIVE
Existing lock ↓				
SHARED	Y	Y	Y	N

Table 4-1. Lock compatibility matrix

| Requested lock ⟶	SHARED	RESERVED	PENDING	EXCLUSIVE
RESERVED	Y	N	N	N
PENDING	N	N	N	N
EXCLUSIVE	N	N	N	N

NOTE

A database system like SQLite needs at least the exclusive lock; the other lock modes just increase transactional concurrency. With only exclusive lock, the system will execute all (read and write) transactions serially. With shared and exclusive locks, the system can execute many read-transactions concurrently. In practice, a transaction reads a data item from a database file under a shared lock, modifies the item, and then requests for an exclusive lock to write the (modified) item back into the file. If two transactions do so simultaneously, there is a possibility of deadlock formation (see the "Deadlock and Starvation" sidebar later in this section), in which the transactions cannot make progress in their executions. The reserved and pending locks are designed to minimize the formation of these kinds of deadlocks. These two locks also help to improve concurrency and to reduce the well-known *writer starvation* problem (in which reads perpetually overtake writes).

Prior to reading the very first (any) page from a database file, a transaction acquires a shared lock that indicates its intention to read pages from the file. Prior to making any changes to a database (at any page), a transaction acquires a reserved lock that indicates its intention to write in the near future. With a reserved lock on a database, a transaction can make changes to "in-cache" pages. Before writing to the database, it needs to obtain an exclusive lock on the database. The locking state transition diagram is given in Figure 4-1.

Figure 4-1. Locking state transition diagram
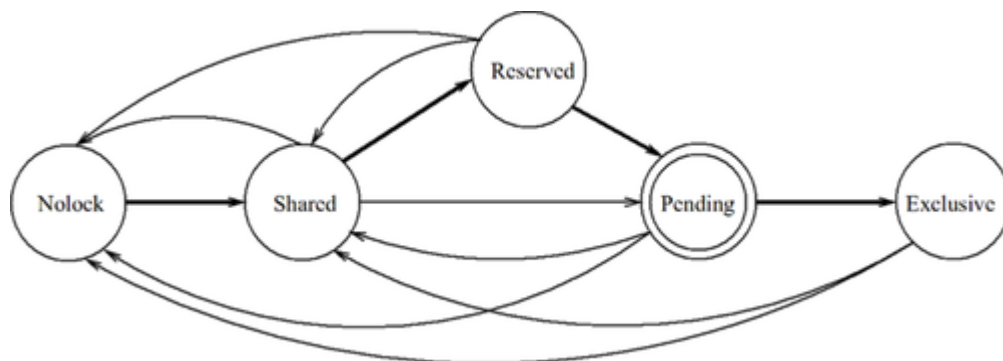
Normal lock transition is from nolock to shared lock to reserved lock to pending. lock to exclusive lock (as shown by bold lines). The direct shared lock to pending lock transition can only happen if there is a journal that needs rolling back. But, if that is the case, then no other transaction could have made the transition from shared lock to reserved lock. (I revisit this in the "Recovery from Failures(See 6.4)" section.)

NOTE

The pending lock is an intermediate lock, and it is not visible outside the lock management subsystem: in particular, the pager cannot ask the lock manager to get a pending lock. It always requests an exclusive lock, but the lock manager first acquires a pending lock en route to getting the exclusive lock. Thus, the pending lock is always just a temporary stepping stone to the path to an exclusive lock. In case the exclusive lock request fails after acquiring a pending lock, the lock will be upgraded to an exclusive lock by a subsequent request for an exclusive lock (from the pager).

NOTE

Although locking solves the concurrency control problem, it introduces another problem. Suppose two transactions hold shared locks on a database file. They both request reserved locks on the file. One of them gets the reserved lock, and the other one waits. After a while, the transaction with the reserved lock requests an exclusive lock, and waits for the other transaction to clear the shared lock. But the shared lock will never be cleared because the transaction with the shared lock is waiting. This kind of situation is called a *deadlock*.

Deadlock is an annoying problem. There are two ways to handle the deadlock problem: (1) prevention, and (2) detection and break. SQLite prevents deadlock formation. It always acquires file locks in nonblocking mode. If it fails to obtain some lock on behalf of a transaction, it will retry only for a finite number of times. (The retry number can be preset by the application at runtime; the default is once.) If all retries fail, SQLite returns SQLITE_BUSY error code to the application. The application can back off and retry later or abort the transaction. Consequently, there is no possibility of deadlock formation in the system. However, there is a possibility, at least theoretically, of starvation where a transaction perpetually tries to obtain some lock without a success. As SQLite is not targeted for enterprise-level highly concurrent applications, starvation is not a big concern.

SQLite implements its own locking system by using the file locking functions supported by the native operating system. (Lock implementation is platform-dependent. I use Linux in this book to show how SQLite locks are implemented.) Linux implements only two lock modes, namely read and write, to lock contiguous regions on files. To avoid confusion in terminologies, I use read lock and write lock for native shared lock and exclusive lock, respectively. Linux allocates locks to processes (in single thread applications) and to threads (in multithreaded applications). To avoid confusion between process and thread, I consistently refer to thread in this subsection.

Many (sibling and peer) threads may hold read locks on a file region, but only one thread may hold a write lock on the region. A write-lock excludes all other locks, both read and write. Read and write locks can coexist on the same file, but at different regions. A single thread can hold only one type of lock on a region. If it applies a new lock to an already-locked region, then the existing lock is converted to the new lock mode.

SQLite implements its own four lock modes using the two native operating system lock modes on different file-regions. (It sets and releases native locks on regions by making `fcntl` system calls.)

- A SHARED lock on a database file is obtained by setting a read lock on a specific range of bytes on the file.
- An EXCLUSIVE lock is obtained by setting a write lock on all bytes in the specified range.
- A RESERVED lock is obtained by setting a write lock on a single byte of the file (that lies outside the shared range of bytes); it is designated as the reserved lock byte.
- A PENDING lock is obtained by setting a write lock on a designated byte different from the reserve lock byte, outside the shared range.

Figure 4-2 displays this arrangement.

Figure 4-2. File offsets that are used to set POSIX locks
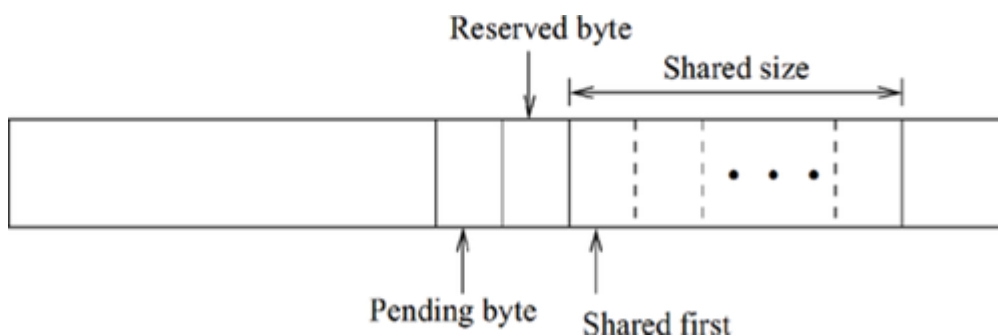
SQLite reserves 510 bytes as the shared range of bytes. (The range value is defined as a macro SHARED_SIZE in a source header file.) The range begins at SHARED_FIRST offset. The PENDING_BYTE macro (0x40000000, the first byte past the 1 GB boundary defines the beginning of the lock bytes) is the byte used for setting PENDING locks. The RESERVED_BYTE macro is set to the next byte after the PENDING_BYTE, and the SHARED_FIRST is set to the second byte after the PENDING_BYTE. All lock bytes will fit into a single database page, even with the minimum page size of 512 bytes.

NOTE

Locking in Windows is mandatory; that is, locks are enforced for all processes, even if they are not cooperating processes. The locking space is reserved by the operating system. For this reason, SQLite cannot store actual data in the locking bytes. Therefore, the pager never

allocates the page involved in locking. That page is also not used in other platforms to have uniformity and cross-platform use of databases. The pending byte is set high so that SQLite doesn't have to allocate an unused page except for very large databases.

To obtain a SHARED lock on a database file, a thread first obtains a native read-lock on the PENDING_BYTE to make sure that no other process/thread has a PENDING lock on the file. (You may recall from Table 4-1 that the existing PENDING lock and a new SHARED lock are incompatible.) If this is successful, the SHARED_SIZE range starting at the SHARED_FIRST byte is read-locked, and, finally, the read-lock on the PENDING_BYTE is released.

NOTE

Some versions of Windows support only write-lock. There, to obtain a SHARED lock on a file, a single byte out of the specific range of bytes is write-locked. The byte is chosen at random from the range so that two separate readers can probably access the file at the same time, unless they are unlucky and choose the same byte to set write-locks. In such systems, read concurrency is limited by the size of the shared range of bytes.

A thread may only obtain a RESERVED lock on a database file after it has obtained a SHARED lock on the file. To obtain a RESERVED lock, a write-lock is obtained on the RESERVED lock byte. Note that the thread does not release its SHARED lock on the file. (This ensures that another thread cannot obtain an EXCLUSIVE lock on the file.)

A thread may only obtain a PENDING lock on a database file after it has obtained a SHARED lock on the file. To obtain a PENDING lock, a write-lock is obtained on the PENDING_BYTE. (This ensures that no new SHARED locks can be obtained on the file, but existing SHARED locks are allowed to coexist.) Note that the thread does not release its SHARED lock on the file. (This ensures that another thread cannot obtain an EXCLUSIVE lock on the file.)

NOTE

A thread does not have to obtain a RESERVED lock on the way to obtaining a PENDING lock. This property is used by SQLite to roll back a journal file after a system crash. If the thread takes the route from shared to reserved to pending, it does not release the other two locks upon setting the pending lock.

A thread may only obtain an EXCLUSIVE lock on a database file after it has obtained a PENDING lock on the file. To obtain an EXCLUSIVE lock, a write-lock is obtained on the entire "shared byte range." Because all other SQLite locks require a read-lock on (at least one of) the bytes within the range, it ensures that no other locks are held on the file when the thread has obtained the exclusive lock.

To have a clear picture on SQLite's way of acquiring native locks on a file, the native locking state transition is drawn in Figure 4-3. The figure also reveals the relationship between SQLite locks and native locks. The representations of PENDING lock and EXCLUSIVE lock are a little

awkward; they may or may not have a write-lock on the RESERVED_BYTE depending on which route SQLite has taken to set those locks.

Figure 4-3. Relationship between SQLite locks and Linux locks
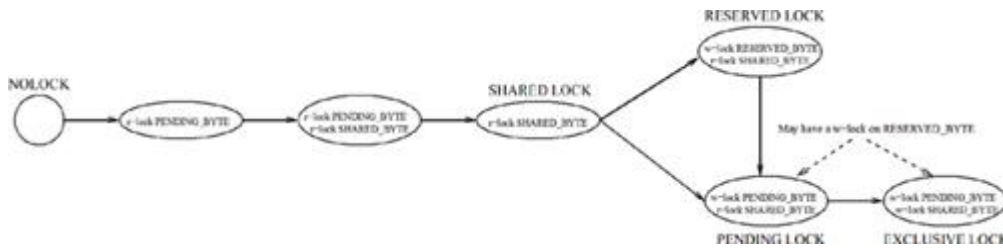
Section 4.3. Journal Management

A *journal* is a repository of information that is used to recover a database when aborting a transaction or statement subtransaction, and also when recovering after an application, system, or power failure. SQLite uses a single journal file per database. (It does not use journal files for in-memory databases.) It assures only rollback (undo, and not redo) of transactions, and the journal file is often called the *rollback journal*. The journal always resides in the same directory as the database file does, and has the same name, but with *-journal* appended.

NOTE

SQLite permits at most one write-transaction on a database file at a time. It creates the journal file on the fly for every write-transaction, and deletes the file when the transaction is complete.

Figure 4-4. Structure of journal segment header

SQLite partitions the rollback journal file into variable size log segments. Each segment starts with a segment header record followed by one or more log records. The format of the segment header is shown in Figure 4-4. The header begins with a magic number containing eight bytes in this sequence: 0xD9, 0xD5, 0x05, 0xF9, 0x20, 0xA1, 0x63, and 0xD7. The magic number is used for a sanity check only, and has no special significance. The number of records (`nRec`, for short) component specifies how many valid log records are in this log segment. The value of the random number component is used to estimate checksums for individual log records. Different segments may have different random numbers. The initial database page count component notes how many pages were in the database file when the current journaling started. The sector size is the size of the disk sector where the journal file resides. The segment header occupies a complete disk sector. The unused space in the header is kept there as filler.

NOTE

SQLite supports asynchronous transactions that are faster than normal transactions. SQLite does not recommend using asynchronous transactions, but you can set the asynchronous mode by executing an SQLite pragma command. This mode is normally used at application development time to reduce the development time. This mode is also satisfactory for some testing applications that do not test for recovery from failures. Asynchronous transactions neither flush the journal nor the database file. The journal file will have only one log segment. The `nRec` value is -1 (i.e., 0xFFFFFFFF as a signed value), and the actual value is derived from the size of the file.

A rollback journal file normally contains a single log segment. But, in some scenarios, it is a multisegment file, and SQLite writes the segment header record multiple times in the file. (You will see such scenarios in the Cache Flush section later.) Each time the header record is written, it is written at the sector boundary. In a multisegment journal file, the `nRec` field in any segment header cannot be 0xFFFFFFFF.

Non-SELECT statements from the current write-transaction produce log records. SQLite uses an old value logging technique at the page level granularity. Before altering any page for the first time, the original content of that page (along with its page number) are written into the journal in a new log record (see Figure 4-5). The record also contains a 32-bit checksum. The checksum covers both the page number and the page image. The 32-bit random value that appears in the log segment header (see the third component in Figure 4-4) is used as the checksum key. The random number is important because garbage data that appears at the end of a journal is likely data that was once in other files that have now been deleted. If the garbage data came from an obsolete journal file, the checksums might be correct. But, by initializing the checksum to a random value that is different for different journal file, SQLite minimizes that risk.

Figure 4–5. Structure of a log record

Page number	Database page image	Checksum
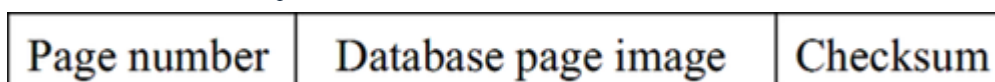
You may recall that an application can attach additional databases to an open connection by executing the SQLite ATTACH command. If a transaction modifies multiple databases, then each database has its own rollback journal. They are independent rollback journals, and not aware of one another. To bridge the gap, SQLite additionally maintains a separate aggregate journal called the *master journal*. The journal does not contain any log records that are used for rollback purposes. Instead, it contains the names of all the individual rollback journals that participate in the transaction. Each individual rollback journal also contains the name of the master journal as shown in Figure 4-6. If there is no attached database, or if no attached database is participating in the current transaction for update purpose, no master journal is

created, and the normal rollback journal does not contain any information about the master journal.

Figure 4-6. Structure of child journal file
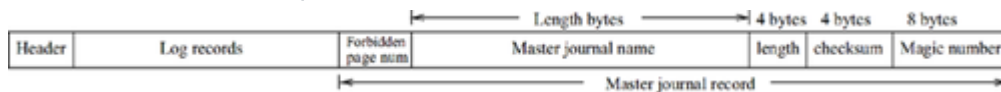
The master journal always resides in the same directory as the main database file does, and has the same name, but with *-mj* appended, followed by eight randomly chosen alphanumeric characters. It is also a transient file. It is created when the transaction attempts to commit, and deleted when the commit processing is complete.

When in a user transaction, SQLite maintains a statement subjournal for the latest non-SELECT statement execution. The journal is required to recover the database from the statement failure. A statement journal is a separate, ordinary rollback journal file. It is an arbitrary named temporary file (prefixed by `sqlite_`). The file is not required for crash recovery operation; it is only needed for statement aborts. SQLite deletes the file when the statement is complete. The journal does not have a segment header record. The `nRec` (number of log records) value is kept in an in-memory data structure, and so is the database file size as of the start of the statement execution. These log records do not have any checksum information.

SQLite follows write-ahead logging (WAL) protocol to ensure durability of database changes, and recoverability of databases in the occurrences of application, system, or power failures. Writing log records in the journal file is lazy: SQLite does not force them to the disk surface immediately. However, before writing the next page in the database file, it forces all log records to disk. This is called *flushing* the journal. Journal flushing is done to make sure all the pages that have been written to the journal have actually reached the disk surface. It is not safe to modify the database file until after the journal has been flushed. If the database is modified before the journal is flushed, and a power failure occurs, the unflushed log records would be lost, and SQLite would not be able to completely roll back the transaction's effects from the database, resulting in database corruption.

The default commit logic is forced-log-at-commit and forced-database-at-commit. When an application commits a transaction, SQLite makes sure that all log records in the rollback journal are in the disk. At the end of the commit, the rollback journal file is deleted, and the transaction is complete. If the system fails before reaching that point, the transaction commit has failed, and it will be rolled back when the database is read next time. However, before deleting the rollback journal file, all changes to the database file are written to the disk. This is done to make

sure that the database has received all updates from the transaction before the journal is deleted.

NOTE

SQLite does not perform journal or database flushing at commit for asynchronous transactions. Thus, upon a failure, the database might be corrupt. Asynchronous transaction writers have been warned!

Section 4.4. Transactional Operations

Like any other DBMS, SQLite's transaction management has two components: (1) normal processing, and (2) recovery processing. During normal processing, the pager saves recovery information in journal files, and it uses the saved information at the recovery processing time if there is a need.

Normal processing involves reading pages from and writing pages into database files, and committing transactions and subtransactions. In addition, the pager flushes the page-cache as a part of normal processing work.

Most transactions and statement subtransactions commit themselves. But occasionally, some transactions or statements abort themselves. In rare cases, there are applications and system failures. In either case, SQLite may need to recover the database into an acceptable consistent state by performing some rollback actions. In the former two cases (statement and transaction aborts), in-memory reliable information might be available at the time of recovery. In the latter case (crashes), the database may be corrupt, and there is no in-memory information.

To act on a database page, the client B^+-tree module needs to apply the `sqlite3pager_get` function on the page number. The client needs to invoke the function, even if the page is nonexistent in the database file: the page will be created by the pager. The function obtains a shared lock on the file if a shared or stronger lock has not already been obtained on the file. If it fails to obtain the shared lock, then some other transaction holds an incompatible lock, and it returns SQLITE_BUSY error code to the caller. Otherwise, it performs a cache read operation (see the "Cache read(See 5.3)" section), and returns the page to the caller. The cache read operation pins down the page.

The first time the pager acquires the shared lock on a database file, we say it has started an implicit read-transaction on the file for the caller. At this point it determines whether or not the file needs a recovery. If the file does need a recovery, the pager performs the recovery before returning the page to the caller. I discuss this in the "Recovery from failures" section.

Before modifying a page, the client B$^+$-tree module must have already pinned down the page (by invoking the `sqlite3pager_get` function). It applies the `sqlite3pager_write` function on the page to make the page writable. The first time the `sqlite3pager_write` function is called on a (any) page, the pager needs to acquire a reserved lock on the database file. If the pager is unable to obtain the lock, it means that another transaction already has a reserved or stronger lock on the file. In that case, the write attempt fails, and the pager returns SQLITE_BUSY to the caller.

The first time the pager acquires a reserved lock, we say it escalates the read-transaction into a write-transaction. At this point, the pager creates and opens the rollback journal. It initializes the first segment header record (see Figure 4-4(See 6.3)), notes down the original size of the database file in the record, and writes the record in the journal file.

To make a page writable, the pager writes the original content of that page (in a new log record) into the rollback journal. Once writable, the client can modify the page as many times as it wants without informing the pager. Changes to the page are not written to the database file immediately.

NOTE

Once a page image is copied into the rollback journal, the page will never be in a new log record, even if the current transaction invokes the `write` function on the page multiple times. One nice property of this logging is that a page can be restored by copying the contents from the journal. Further, the undo is idempotent, and hence undos do not produce any compensating log records.

SQLite never saves a new page (that is added, i.e., appended, to the database file by the current transaction) in the journal because there is no old value for the page. Instead, the journal notes the initial size of the database file in the journal segment header record (see Figure 4-4(See 6.3)) when the journal file is created. If the database file is expanded by the transaction, the file will be truncated to its original size on a rollback.

The cache flush is an internal operation of the pager module; the module client can never invoke a cache flush directly. Eventually, the pager wants to write some modified pages back to the database file, either because the cache has filled up and there is a need for cache replacement, or because the transaction is ready to commit its changes. The pager performs the following steps:

1. It determines whether there is a need to flush the journal file. If the transaction is not asynchronous, and has written new data in the journal file, and the database is not a temporary file,[*] then the pager decides to do a journal flush. In that case, it makes a `fsync` system call on the journal file to ensure that all log records written so far have actually reached the surface of the disk (and are not just being

held in the operating system space or the disk controller's internal memory). At this time, the pager does not write the number of log records (`nRec`) value in the current log segment header. (The `nRec` value is a precious resource for rollback operation. When the segment header is formed, the number is set to zero for synchronous transactions and to 0xFFFFFFFF for asynchronous ones.) After the journal is flushed, the pager writes the `nRec` value in the current log segment header, and flushes the file again.[†] As disk writes are not atomic, it will not rewrite the `nRec` field any more. It instead creates a new log segment for the new oncoming log records. In these scenarios, SQLite uses multisegment journal files.

[*] For temporary databases, we do not care if we are able to roll back after a power failure, so no journal flush occurs.

[†] The journal file is flushed twice. The second flush leads to an overwrite of the disk block that stores the `nRec` field. If we assume that this overwrite is atomic, then we are guaranteed that the journal will not be corrupted at this point of flushing. Otherwise, we are at some minor risk.

2. It tries to obtain an EXCLUSIVE lock on the database file. (If other transactions are still holding SHARED locks, the lock attempt fails, and it returns SQLITE_BUSY to the caller. The transaction is not aborted.)
3. It writes all modified pages or selective ones out to the database file. The page writing is done in-place. It marks cache copies of these pages clean. (It does not flush the database file to disk at this time.)

If the reason for writing to the database file is because the page cache is full, then the pager does not commit the transaction right away. Instead, the transaction might continue to make changes to other pages. Before subsequent changes are written to the database file, these three steps are repeated once again.

NOTE

The EXCLUSIVE lock the pager has obtained to write to the database file is held until the transaction is complete. It means that other applications will not be able to open another (read or write) transaction on the database from the time the pager first writes the database file until the transaction commits or aborts. For short transactions, updates are held in-cache, and the exclusive lock will be acquired only at the commit time for the duration of the commit.

SQLite follows a slightly different commit protocol depending on whether the committing transaction modifies a single database or multiple databases.

4.4.4.1. Single database case

Committing a read-transaction is easy. The pager releases the shared lock from the database file, and purges the page cache. To commit a write-transaction, the pager performs the following steps in the order listed:

1. It obtains an EXCLUSIVE lock on the database file. (If the lock acquisition fails, it returns SQLITE_BUSY to the caller. It cannot commit the transaction now, as other transactions are still reading the database.) It writes all modified pages back to the database file following the algorithmic steps 1 through 3 given in the "Cache flush" section.
2. The pager makes an `fsync` system call on the database file to flush the file to the disk.
3. It then deletes the journal file.
4. Finally, it releases the EXCLUSIVE lock from the database file, and purges the page cache.

NOTE

The transaction commit point occurs at the instant the rollback journal file is deleted. Prior to that, if a power failure or crash occurs, the transaction is considered to have failed during the commit processing. The next time SQLite reads the database, it will roll back the transaction's effects from the database.

4.4.4.2. Multidatabase case

The commit protocol is a little more involved, and it resembles a transaction commit in distributed systems. The VM (virtual machine) module actually drives the commit protocol as the commit coordinator. Each pager does its own part of "local" commit on its database. For a read-transaction or a write-transaction that modifies a single database file (the `temp` database is not counted), the protocol executes a normal commit on each database involved. If the transaction modifies more than one database file, the following commit protocol is performed:

1. Release the SHARED lock from those databases the transaction did not update.
2. Acquire EXCLUSIVE locks on those databases the transaction has updated.
3. Create a new master journal file. Fill the master journal with the names of all individual rollback journal files, and flush the master journal and the journal directory to disk. (The temp database name is not included in the master journal.)
4. Write the name of the master journal file into all individual rollback journals in a master journal record (see Figure 4-6(See 6.3), earlier), and flush the rollback journals. (A pager may not know that it has been a part of multidatabase transaction until the transaction commit time. Only at this point it comes to know that it is a part of a multidatabase transaction.)
5. Flush individual database files.
6. Delete the master journal file and flush the journal directory.
7. Delete all individual rollback journal files.
8. Release EXCLUSIVE locks from all database files, and purge those page-caches.

NOTE

The transaction is considered to have been committed when the master journal file is deleted. Prior to that, if a power failure or crash occurs, the transaction is considered to have failed during the commit processing. When SQLite reads these databases next time, it will recover them to their respective states prior to the start of the transaction.

> If the main database is a temporary file (or in-memory), SQLite does not guarantee atomicity of the multidatabase transaction. That is, the global recovery may not be possible. It does not create a master journal. The VM follows the simple commit on individual database files, one after another. The transaction is thus guaranteed to be locally atomic within each individual database file. Thus, on occurrence of a failure, some of those databases might get the changes and some might not.

Normal operations at statement subtransaction level are read, write, and commit. These are discussed below.

4.4.5.1. Read operation

A statement subtransaction reads pages through the main user transaction. All rules are followed as those for the main transaction.

4.4.5.2. Write operation

There are two parts in a write operation: locking and logging. A statement subtransaction acquires locks through the main user transaction. But statement logging is a little different, and is handled by using a separate temporary statement journal file. SQLite writes some log records in the statement journal, and some in the main rollback journal. The pager performs one of the following two alternative actions when a subtransaction tries to make a page writable via the `sqlite3pager_write` operation:

1. If the page is not already in the rollback journal, it adds a new log record to the rollback journal.
2. Otherwise, it adds a new log record to the statement journal if the page is not already in this journal. (The pager creates the journal file when the transaction writes the first log record in the file.)

The pager never flushes a statement journal, because this is never required for failure recovery. If a power loss occurs, the main rollback journal will take care of database recovery. You may note that an in-cache page can be a part of both the rollback journal and the statement journal: the rollback journal has the oldest page image.

4.4.5.3. Commit operation

Statement commit is very simple. The pager deletes the statement journal file.

Recovery from abort is very simple in SQLite. The pager may or may not need to remove the effects of the transaction from the database file. If the transaction holds only a RESERVED or PENDING lock on the database, it is guaranteed that the file is not modified; the pager deletes the journal file, and discards all dirty pages from the page cache. Otherwise, some pages are written back in the database file by the transaction, and the pager performs the following rollback actions:

The pager reads log records one after another from the rollback journal file, and restores the page images from the records. (You may recall that a database page is logged at most once by the transaction and the log record stores the before image of the page.) Thus, at the end of the journal scan, the database is restored to its original state prior to the start of the transaction. If the transaction has expanded the database, the pager truncates the database to the original size. It (the pager) then flushes the database file first and deletes the rollback journal file next. It releases the exclusive lock, and purges the page cache.

As noted in the "Statement operations" section, a statement may have added log records to both the rollback journal and the statement journal. SQLite needs to roll back all log records from the statement journal, and some from the rollback journal. When a statement subtransaction writes the first log record in the rollback journal, the pager keeps a note of the record position in an in-memory data structure. This record onward, until the end of the rollback journal file, is written by the subtransaction. The pager restores the page images from those log records. It then deletes the statement journal file, but keeps the rollback journal file without changing the content. When a statement subtransaction starts, the pager also records the database size. The pager truncates the database file to that size.

After a crash or system failure, inconsistent data may have been left in a database file. When no application is updating a database, but there is the rollback journal file, it means that the previous transaction may have failed, and SQLite may need to recover the database from the effects of the transaction before the database can be used for normal business. A rollback journal file is said to be *hot* if the corresponding database file is unlocked or shared locked. A journal becomes hot when a write-transaction is midway toward its completion and a failure prevents the completion. However, a rollback journal is not hot if it is produced by a multidatabase transaction and there is no master journal file; this implies that the transaction is committed by the time the failure had occurred. A hot journal implies that it needs to be rolled back to restore the database consistency.

NOTE

If no master journal is involved, then a rollback journal is hot if it exists and the database file does not have a reserved or stronger lock. (You may recall that a transaction with a reserved lock creates a rollback journal file; this file is not hot.) If a master journal name appears in the rollback journal, then the rollback journal is hot if the master journal exists and there is no reserved lock on the corresponding database file.

When a database starts, in most DBMSs, the transaction manager initiates a recovery operation on the database immediately. SQLite uses a deferred recovery. As explained in the "Read operation" section, when the first read of a (any) page from the database is performed, the pager goes through the recovery logic and recovers the database only if the rollback journal is hot.

> If the current application only has the read permission on the database file, and no write permission on the file or on the containing directory, the recovery fails, and the application gets an unexpected error code from the SQLite library.

It performs the following sequence of recovery steps before it actually reads a page from the file:

1. It obtains a SHARED lock on the database file. If it cannot get the lock, it returns SQLITE_BUSY to the application.
2. It checks if the database has a hot journal. If the database does not have a hot journal, the recovery operation finishes. If there is a hot journal, the journal is rolled back by the subsequent steps of this algorithm.
3. It acquires an EXCLUSIVE lock (via a PENDING lock) on the database file (see Figure 4-1(See 6.2)). (The pager does not acquire a RESERVED lock because that would make other pagers think the journal is no longer hot and read the database. It needs an exclusive lock because it is about to write to the database file as a part of the recovery work.) If it fails to acquire these locks, it means another pager is already trying to do the rollback. In that case, it drops all locks, closes the database file, and returns SQLITE_BUSY to the application.
4. It reads all log records from the journal file and undoes them. This restores the database to its original state prior to the start of the crashed transaction, and hence, the database is in a consistent state now.
5. It flushes the database file. This protects the integrity of the database in case another power failure or crash occurs.
6. It deletes the rollback journal file.
7. It deletes the master journal file if it is safe to do so. (This step is optional. See the next sidebar for details.)

8. It drops the EXCLUSIVE (and PENDING) locks, but retains the SHARED lock. (This is because the pager performs the recovery in the `sqlite3pager_get` function.)

After the preceding algorithm completes successfully, the database file is guaranteed to have been restored to the state as of the start of the failed transaction. It is safe to read from the file now.

NOTE

A master journal is *stale* if no individual rollback journal is pointing to it. To figure out whether a master journal is stale, the pager first reads the master journal to obtain the names of all rollback journals. It then examines each of those rollback journals. If any of them exists and points back to the master journal, then the master journal is not stale. If all rollback journals are either missing or refer to other master journals or to no master journal at all, then the master journal is stale, and the pager deletes the journal. There is no requirement that stale master journals be deleted. The only reason for doing so is to free up disk space occupied by them.

To reduce workload at failure recovery time, most DBMSs perform checkpoints on database at regular intervals. You may recall that SQLite can have at most one write-transaction on a database file at a time. The journal file contains log records from only that transaction, and SQLite deletes the journal file when the transaction completes. Consequently, SQLite does not need to perform checkpoints, and it does not have any checkpoint logic embedded in it. When a transaction commits, SQLite makes sure that all updates from the transaction are in the database file before deleting the journal file.

Chapter 5. Table and Index Management

I already discussed how the pager module implements a page-oriented file abstraction on the top of a native byte-oriented file. In this section, I will discuss how the B$^+$-tree module implements a tuple (row) oriented file abstraction on the top of a page-oriented file.

Tuples of a table can be organized in many ways, such as entry sequence, relative, hash, key sequence. SQLite uses a single B$^+$-tree to organize all tuples of a table, and different ones for different tables. SQLite treats the content of an index as a kind of table, and stores the content in a B-tree and different indexes in different B-trees. B- and B$^+$-trees are similar kind of key sequence data structures. SQLite does not use any other tuple organization techniques. Consequently, a database is a collection of B- and B$^+$-trees. All of these trees are spread across database pages, and they can be interspersed. But, no database page stores tuples from two or more tables or indexes. It is the duty of the tree module to organize pages of a tree so that tuples can be stored and retrieved efficiently.

In the rest of this section, I restrict myself primarily to B$^+$-tree. I will use B$^+$-tree for generic purpose. The module implements primitives to read, insert, and delete individual tuples from trees, and, of course, to create and delete trees. It is a passive entity as far as the structures of tuples are concerned, and it treats tuples as variable length byte strings.

5.1. B$^+$-Tree Structure

B-tree, where B stands for "balanced," is by far the most important index structure in many external storage based DBMSs. It is a way of organizing a collection of similar data records in a sorted order by their keys. (A *sort order* is any total order on the keys. Different B-trees in the same database can have different sort orders.) B-tree is a special kind of height balanced n-ary, $n > 2$, tree, where all leaf nodes are at the same level. Entries[*] and search information (i.e., key values) are stored in both internal nodes and leaf nodes. B-tree provides near optimum performance over the full range of tree operations, namely insertion, deletion, search, and search-next.

[*] To avoid confusion, I use the term "entry" for a tuple or record here. An entry consists of a key and other optional data.
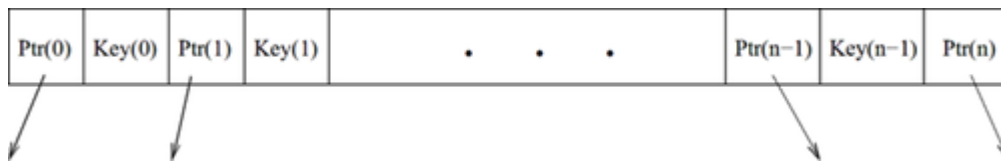
A B$^+$-*tree* is a variant of B-tree, where all entries reside at leaf nodes. The entries are (key value, data value) pairs; and they are sorted by the key values. Internal nodes contain only search information (key values) and child pointers. Keys in an internal node are stored in the sort order, and are used in directing a search to appropriate child node.

For either kind of trees, internal nodes can have a variable number of child pointers within a preset range. For a particular implementation, there are lower and upper bounds on the number of child pointers an internal node can have. The lower bound is normally larger or equal to half of the upper bound. The root node can violate this rule; it can have any number of child pointers (from zero up to the upper bound limit). All leaf nodes are at the same lowest level, and in some implementations, they are linked in an ordered chain. The root node is always an internal node for B$^+$-tree.

For a given upper bound of $n + 1$, $n > 1$, in B$^+$-trees, each internal node contains at most n keys and at most $n + 1$ child pointers. The logical organization of keys and pointers is shown in Figure 5-1. For any internal node, the following holds:

- All of the keys on the child subtree that `Ptr(0)` points to have values less than or equal to `Key(0)`;
- All of the keys on the child subtree that `Ptr(1)` points to have values greater than `Key(0)` and less than or equal to `Key(1)`, and so forth;
- All of the keys on the child subtree that `Ptr(n)` points to have values greater than `Key(n - 1)`.

Finding a particular entry for a given key value requires traversing `O(log` m`)` nodes, where m is the total number of entries in the tree.

Figure 5-1. Structure of a B$^+$-tree internal node

Section 5.1. B+-Tree Structure

Chapter 5. Table and Index Management

I already discussed how the pager module implements a page-oriented file abstraction on the top of a native byte-oriented file. In this section, I will discuss how the B$^+$-tree module implements a tuple (row) oriented file abstraction on the top of a page-oriented file.

Tuples of a table can be organized in many ways, such as entry sequence, relative, hash, key sequence. SQLite uses a single B$^+$-tree to organize all tuples of a table, and different ones for different tables. SQLite treats the content of an index as a kind of table, and stores the content in a B-tree and different indexes in different B-trees. B- and B$^+$-trees are similar kind of key sequence data structures. SQLite does not use any other tuple organization techniques. Consequently, a database is a collection of B- and B$^+$-trees. All of these trees are spread across database pages, and they can be interspersed. But, no database page stores tuples from two or more tables or indexes. It is the duty of the tree module to organize pages of a tree so that tuples can be stored and retrieved efficiently.

In the rest of this section, I restrict myself primarily to B$^+$-tree. I will use B$^+$-tree for generic purpose. The module implements primitives to read, insert, and delete individual tuples from trees, and, of course, to create and delete trees. It is a passive entity as far as the structures of tuples are concerned, and it treats tuples as variable length byte strings.

5.1. B$^+$-Tree Structure

B-tree, where B stands for "balanced," is by far the most important index structure in many external storage based DBMSs. It is a way of organizing a collection of similar data records in a sorted order by their keys. (A *sort order* is any total order on the keys. Different B-trees in the same database can have different sort orders.) B-tree is a special kind of height balanced n-ary, $n > 2$, tree, where all leaf nodes are at the same level. Entries[*] and search information (i.e., key values) are stored in both internal nodes and leaf nodes. B-tree provides near optimum performance over the full range of tree operations, namely insertion, deletion, search, and search-next.

[*] To avoid confusion, I use the term "entry" for a tuple or record here. An entry consists of a key and other optional data.

A B$^+$-*tree* is a variant of B-tree, where all entries reside at leaf nodes. The entries are (key value, data value) pairs; and they are sorted by the key values. Internal nodes contain only search

information (key values) and child pointers. Keys in an internal node are stored in the sort order, and are used in directing a search to appropriate child node.
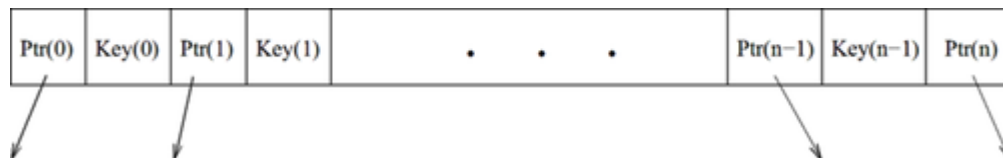
For either kind of trees, internal nodes can have a variable number of child pointers within a preset range. For a particular implementation, there are lower and upper bounds on the number of child pointers an internal node can have. The lower bound is normally larger or equal to half of the upper bound. The root node can violate this rule; it can have any number of child pointers (from zero up to the upper bound limit). All leaf nodes are at the same lowest level, and in some implementations, they are linked in an ordered chain. The root node is always an internal node for B$^+$-tree.

For a given upper bound of $n + 1$, $n > 1$, in B$^+$-trees, each internal node contains at most n keys and at most $n + 1$ child pointers. The logical organization of keys and pointers is shown in Figure 5-1. For any internal node, the following holds:

- All of the keys on the child subtree that `Ptr(0)` points to have values less than or equal to `Key(0)`;
- All of the keys on the child subtree that `Ptr(1)` points to have values greater than `Key(0)` and less than or equal to `Key(1)`, and so forth;
- All of the keys on the child subtree that `Ptr(n)` points to have values greater than `Key(n - 1)`.

Finding a particular entry for a given key value requires traversing `O(log` m`)` nodes, where m is the total number of entries in the tree.
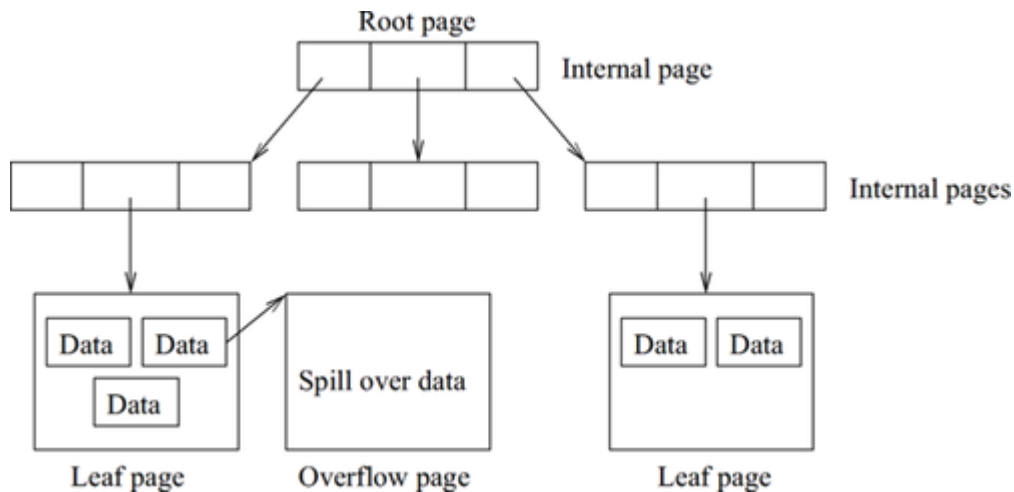
Figure 5-1. Structure of a B$^+$-tree internal node

Section 5.2. B+-tree in SQLite

A tree is created by allocating its root page. The root page is not relocated. Each tree is identified by its root page number. The number is stored in the master catalog table, whose root always resides on Page 1.

Figure 5-2. A typical B$^+$-tree

SQLite stores tree nodes (both internal and leaf) in separate pages—one node in one page (see Figure 5-2).The pages are accordingly referred to as internal and leaf pages, respectively. For each node, the key and data for any entry are combined to form a *payload*. A fixed preset amount of payload is stored directly on the page. If the payload is larger than the amount, then surplus bytes are spilled into overflow pages: excess payload is sequentially stored in a linked list of overflow pages. Internal nodes, not shown in the figure, can have overflow pages, too.

Section 5.3. Page Structure

You may recall that each database file is divided into fixed size pages. The B^+-tree module manages all those pages. Each page is either a tree page (internal or leaf), or an overflow page, or a free page. (Figure 2-1(See 4.2) in the "Database File Structure(See 4.2)" section shows that free pages are organized into a single trunk list.) In this section, we study structures of internal, leaf, and overflow pages.

The logical content of each tree page is partitioned into what are called *cells:* a cell contains a (part of a) payload. Cells are units of space allocation and deallocation on tree pages. The format of a tree page is given in Figure 5-3.

Figure 5-3. Structure of a tree page
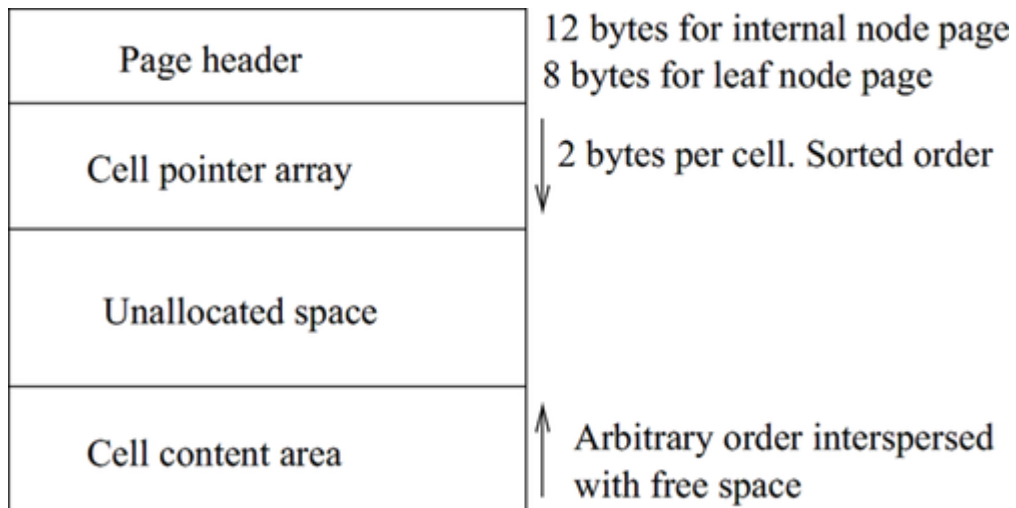
Each tree page is divided into four sections:

1. The page header
2. The cell content area
3. The cell pointer array
4. Unallocated space

The cell pointer array and the cell content area grow toward each other. The cell pointer array acts as the page-directory that helps map logical cell order to their physical cell storage in the cell content area.

A page header contains the management information only for that page. The header is always stored at the beginning of page (low address). (Page 1 is an exception: the first 100 bytes contain the file header.) The structure of the page header is given in Table 5-1. The first two columns are in bytes.

Table 5-1. Structure of tree page header

Offset	Size	Description
0	1	Flags. 1: intkey, 2: zerodata, 4: leafdata, 8: leaf
1	2	Byte offset to the first free block
3	2	Number of cells on this page
5	2	First byte of the cell content area
7	1	Number of fragmented free bytes
8	4	Right child (the `Ptr(n)` value). Omitted on leaves.

The flags at offset 0 define the format of the page. If the `leaf` flag bit is true, it means that the page is a leaf node and has no children. If the `zerodata` flag bit is true, it means that this page carries only key values and no data. If the `intkey` flag bit is true, it means that the key is an integer that is stored in the key size entry in the cell header rather than in the payload area (see later in this section). If the `leafdata` flag bit is true, it means that the tree stores data on leaf nodes only. For internal pages, the header also contains the rightmost child pointer at offset 8.

Cells are stored at the very end of the page (high address), and they grow toward the beginning of the page. The cell pointer array begins on the first byte after the page header, and it contains zero or more cell pointers. Each cell pointer is a 2-byte integer number indicating an offset (from the beginning of the page) to the actual cell within the cell content area. The cell pointers are stored in sorted order (by the corresponding key values), even though cells may be stored unordered. The number of elements in the array is stored in the page header at offset 3.

Because of random inserts and deletes of cells, a page may have cells and free space interspersed (inside the cell content area). The unused space within the cell content area is collected into a linked list of free blocks. The blocks on the list are arranged in ascending order of their addresses. The head pointer (a 2-byte offset) to the list originates in the page header at offset 1. Each free block is at least 4 bytes in size. The first 4 bytes on each free block store control information: the first 2 bytes for the pointer to the next free block (a zero value indicates no next free block), and the other 2 bytes for the size of this free block. Because a free block must be at least 4 bytes in size, any group of 3 or fewer unused bytes (called a *fragment*) in the cell content area cannot exist on the free block chain. The cumulative size of all fragments is recorded in the page header at offset 7. (The size can be at most 255. Before it reaches the max value, the page is defragmented.) The first byte of the cell content area is recorded in the page header at offset 5. The value acts as the border between the cell content area and the unallocated space.

Cells are variable length byte strings. A cell stores a single payload. The structure of a cell is given in Table 5-2. The size column is in bytes.

Table 5-2. Structure of a cell

Size	Description
4	Page number of the left child. Omitted if `leaf` flag bit is set.
var(1–9)	Number of bytes of data. Omitted if the `zerodata` flag bit is set.
var(1–9)	Number of bytes of key. Or the key itself if `intkey` flag bit is set.
*	Payload
4	First page of the overflow chain. Omitted if no overflow.

For internal pages, each cell contains a 4-byte child pointer; for leaf pages, cells do not have child pointers. Next comes the number of bytes in the data image, and the number of bytes in

the key image. (The key image does not exist if the `intkey` flag bit is set in the page header. The key length bytes instead store the integer key value. The data image is nonexistent if the `zerodata` flag is set in the page header.) Figure 5-4 depicts the cell format: (a) structure of a cell, and (b) structure of payloads. The key or the data or both may be absent in a payload.

Figure 5-4. Cell organization

(a) Structure of cell

(b) Structure of payload
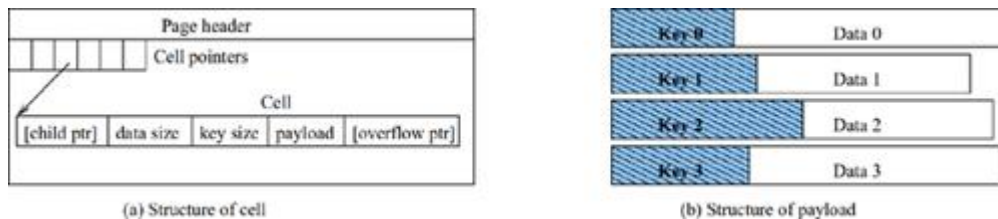
SQLite uses variable length integers to represent integer sizes (and integer key values). A variable length integer is 1 to 9 bytes long, where the lower 7 bits of each byte are used for the integer value calculation. The integer consists of all consecutive bytes that have bit 8 set and the first byte with bit 8 clear. The most significant byte of the integer appears first. A variable-length integer is at most 9 bytes long. As a special case, all 8 bits of the 9th byte are used in the value determination. This representation allows SQLite to encode a 64-bit integer in at most 9 bytes. It is called the Huffman code, and it was invented in 1952. SQLite prefers the variable length Huffman encoding over fixed length 64-bit encoding because only one or two bytes are required to store the most common cases, but up to 64-bits of information can he encoded (in 9 bytes) if needed.

You may recall that SQLite puts a restriction in storing payloads on a tree page. A payload might not be stored in its entirety on a page even if there is sufficient free space on the page. The maximum embedded payload fraction is the amount of the total usable space in a page that can be consumed by a single cell of an internal page. This value is available in the database file header at offset 21 (see Table 2-1(See 4.2) in the "Database File Structure(See 4.2)" section). If the payload of a cell on an internal page is larger than the max payload, then extra payload is spilled into a chain of overflow pages. Once an overflow page is allocated, as many bytes as possible are moved into the overflow pages without letting the cell size drop below the min embedded payload fraction value (in the database file header at offset 22). The min leaf payload fraction (in the file header at offset 23) is like the min embedded payload fraction, except that it applies to leaf pages. The maximum payload fraction for a leaf page is always 100 percent, and it is not specified in the header.

Multiple small entries can fit on a single tree page, but a large entry can span multiple overflow pages. Overflow pages (for a cell) form a singly linked list. Each overflow page except the last one is completely filled with data of length equal to usable space minus four bytes: the first four bytes store the next overflow page number. The last overflow page can have as little as one byte of data. An overflow page never stores content from two cells.

Section 5.4. Module Functionality

The module helps the VM to organize all tables and indexes into B/B$^+$-trees: one B$^+$-tree for each table, and one B-tree for each index. Each tree consists of one or more database pages. The VM can store and retrieve variable length records in any tree. It can delete records from a tree any time. The tree module does self-balancing on insert or delete, and it performs automatic reclamation and reuse of free space. For a tree with m tuples, the module provides the VM with O(log m) time-bound lookup, insert, and delete of records, and O(1) amortized bidirectional traversal of records. For more information on B- and B$^+$-tree algorithms, see "The Art of Computer Programming, Volume 3: Sorting and Searching" by Donald E. Knuth.

The module receives requests for cell insertions and deletions at random from the VM. An insert operation requires allocation of space in tree (and overflow) pages. A delete operation frees up occupied space from tree (and overflow) pages. The management of free space on each page is very crucial for effective utilization of space allocated to the database.

5.4.1.1. Management of free pages

When a page is removed from a tree, the page is added to the file freelist for later reuse. When a tree needs expansion, a page is taken off the freelist and is added to the tree. If the freelist is empty, a page is taken from the native filesystem. (Pages taken from the filesystem are always appended at the end of the database file.)

5.4.1.2. Management of page space

There are three partitions of free space on a tree page:

1. The space between the cell pointer array and the top of the cell content area.
2. The free blocks inside the cell content area.
3. Scattered fragments in the cell content area.

The space allocation is done from the former two partitions. At each allocation or deallocation, the corresponding affected partition is updated accordingly:

Cell allocation

The space allocator does not allocate space less than 4 bytes; such requests are rounded up to 4 bytes. Suppose a new request for `nRequired`, `nRequired >= 4`, bytes comes on a page when the page has is a total of `nFree` bytes. If `nRequired > nFree`, the request fails. Suppose `nRequired <= nFree`. The allocator performs the following steps to satisfy the request:

1. It traverses down the free block list to see whether there is a large enough block to satisfy the request. This is a first fit search. If a suitable block is found, it does one of the following:
   a. If the block size is less than `nRequired + 4`, it removes the block from the free block list, satisfies the request from the beginning of the block, and puts the remaining space (<= 3 bytes) in the fragment partition.
   b. Otherwise, it satisfies the request from the bottom part of the block, and reduces the size of the block by `nRequired` bytes.
2. Otherwise, there is no sufficiently large free block to satisfy the request. If there is not much space in the middle unallocated partition, or there are too many fragments, the module defragments the page first. It runs a compaction algorithm on the page to consolidate the entire free space in the middle. During the compaction, it transfers the existing cells, one after another, to the bottom of the page.
3. It allocates `nRequired` bytes from the bottom of the free space area, and increases the top value by `nRequired` bytes.

Cell deallocation

Suppose a request comes to release `nFree(>=` bytes that were previously allocated by the allocator. The allocator creates a new free block of size `nFree` bytes, and inserts the block in the free block list at appropriate position. Then, it tries to merge free blocks in the neighborhood of the released one. If there is a fragment in between two adjacent free blocks, it also merges the fragment with the blocks. If there is a free block right at the top pointer, it merges the block with the space in the middle unallocated partition, and increases the value of the top.

Chapter 6. SQLite Engine

The topmost module of the backend is popularly called virtual database engine, or the *virtual machine* (VM) in SQLite terminology. The VM is the heart of SQLite, and is the interface between the frontend and the backend. Core information processing happens in it. It implements an abstraction of a new machine on the top of the native system, and it executes programs written in the SQLite internal bytecode programming language. This programming language is specifically designed to search, read, and modify databases. The VM accepts bytecode programs (generated by the frontend), and executes the programs. (You may recall that bytecode programs are prepared statements.) The VM uses the infrastructures provided by the B$^+$-tree module to execute bytecode programs and to produce output of program execution.

The VM does not do any query optimization work. It blindly executes bytecode programs. In doing so, it converts data from one format into another on demand. On-the-fly-data conversion is the primary task of the VM; everything else is controlled by the bytecode programs it executes.

A bytecode program is encapsulated by an in-memory object of type `sqlite3_stmt` (internally called `Vdbe`). Following SQLite, APIs can be applied on the object to manipulate it, to execute the bytecode program, and to retrieve output produced by the program: `sqlite3_bind_*`, `sqlite3_step`, `sqlite3_column_*`, `sqlite3_finalize`.

The internal state of a `Vdbe` object includes the following:

- a bytecode program
- names and data types for all result columns
- values bound to input parameters
- a program counter
- an execution stack of operands
- an arbitrary amount of "numbered" memory cells
- other run-time state information (such as open `BTree` objects, sorters, lists, sets)

6.1. Bytecode Programming Language

SQLite defines an internal programming language to prepare bytecode programs. The language is akin to the assembly language used by physical as well as virtual machines: it defines bytecode instructions. A bytecode instruction is of the form `<opcode, P1, P2, P3>`, where `opcode` identifies a specific bytecode operation, and P1, P2, and P3 are operands to the operation. Each bytecode operation defines a small amount of VM work. The P1 operand is a 32-bit signed integer. The P2 operand is a 31-bit non-negative integer; it is always the jump destination in any operation that might cause a jump. It is also used for other purposes. The P3 operand is a pointer to a null terminated string, or a pointer to a different structured object or a native NULL (0). Some opcodes use all three operands, some typically ignore one or two operands, and many ignore all three operands.

NOTE

Opcodes are internal VM operation names, and they are not a part of the SQLite interface specification. Consequently, their operational semantics may change from one release to another. The SQLite development team does not encourage SQLite users to write bytecode programs on their own.

Table 6-1 displays a typical bytecode program that is equivalent to `SELECT * FROM t1`. The table `t1` has two columns, namely `x` and `y`. The top line on the table is not a part of the program. Every other line is a bytecode instruction.

Table 6-1. A typical bytecode program

Address	Opcode	P1	P2	P3
0	Goto	0	11	
1	Integer	0	0	
2	OpenRead	0	2	#t1
3	SetNumColumn	0	2	
4	Rewind	0	9	
5	Column	0	0	#x
6	Column	0	1	#y
7	Callback	2	0	
8	Next	0	5	
9	Close	0	0	
10	Halt	0	0	
11	Transaction	0	0	
12	VerifyCookie	0	1	
13	Goto	0	1	

The VM is an interpreter, and here is its structure:

```
for (; pc < nOp && rc == SQLITE_OK; pc++){
  switch (aOp[pc].opcode){
  case OP_Add:
    /* Implementation of the ADD operation here */
    break;
  case OP_Goto:
    pc = op[pc].p2-1;
    break;
  case OP_Halt:
    pc = nOp;
    break;
  /* other cases for other opcodes */
  }
}
```

The interpreter is a simple loop containing a massive switch statement. Each case statement implements one bytecode instruction. (Opcode names are prefixed by `OP_`.) In each iteration, the VM fetches the next bytecode instruction from the program, i.e., from `aOp` array using `pc` (both are members of `Vdbe` object) as index into the array. It decodes and carries out the operation specified by the instruction. The VM begins an execution of a bytecode program starting at the instruction number 0.

The VM accesses a database using cursors. It can have zero or more open cursors on the database. Each cursor is a pointer into a single table or index tree. The cursor can seek to an entry with a particular key, or loop over all entries of the tree. The VM inserts new entries, retrieves the key/data at the current entry on the cursor, or deletes the entry.

The VM uses an operand stack and an arbitrary amount of numbered memory locations to hold all intermediate results. Many of the opcodes use operands from the stack. Computation results are also stored on the stack. Each stack or memory location holds a single data value. The memory locations are typically used to hold the result of a scalar SELECT that is part of a larger expression.

The VM continues a bytecode program execution until it processes a halt instruction or encounters an error (in the interpreter program, the `rc` variable stores the status of instruction executions), or the program counter points past the last instruction. When the VM halts, it releases all allocated memory, and closes all cursors. If the execution has stopped due to an error, the VM terminates the transaction or subtransaction, and removes changes made by the (sub)transaction from the database.

Section 6.1. Bytecode Programming Language

Chapter 6. SQLite Engine

The topmost module of the backend is popularly called virtual database engine, or the *virtual machine* (VM) in SQLite terminology. The VM is the heart of SQLite, and is the interface between the frontend and the backend. Core information processing happens in it. It implements an abstraction of a new machine on the top of the native system, and it executes programs written in the SQLite internal bytecode programming language. This programming language is specifically designed to search, read, and modify databases. The VM accepts bytecode programs (generated by the frontend), and executes the programs. (You may recall that bytecode programs are prepared statements.) The VM uses the infrastructures provided by the B$^+$-tree module to execute bytecode programs and to produce output of program execution.

The VM does not do any query optimization work. It blindly executes bytecode programs. In doing so, it converts data from one format into another on demand. On-the-fly-data conversion is the primary task of the VM; everything else is controlled by the bytecode programs it executes.

A bytecode program is encapsulated by an in-memory object of type `sqlite3_stmt` (internally called `Vdbe`). Following SQLite, APIs can be applied on the object to manipulate it, to execute the bytecode program, and to retrieve output produced by the program: `sqlite3_bind_*`, `sqlite3_step`, `sqlite3_column_*`, `sqlite3_finalize`.

The internal state of a `Vdbe` object includes the following:

- a bytecode program
- names and data types for all result columns
- values bound to input parameters
- a program counter
- an execution stack of operands
- an arbitrary amount of "numbered" memory cells
- other run-time state information (such as open `BTree` objects, sorters, lists, sets)

6.1. Bytecode Programming Language

SQLite defines an internal programming language to prepare bytecode programs. The language is akin to the assembly language used by physical as well as virtual machines: it defines bytecode instructions. A bytecode instruction is of the form <`opcode, P1, P2, P3`>, where `opcode` identifies a specific bytecode operation, and P1, P2, and P3 are operands to the operation. Each bytecode operation defines a small amount of VM work. The P1 operand is a 32-bit signed integer. The P2 operand is a 31-bit non-negative integer; it is always the jump destination in any operation that might cause a jump. It is also used for other purposes. The P3 operand is a pointer to a null terminated string, or a pointer to a different structured object or a native NULL (0). Some opcodes use all three operands, some typically ignore one or two operands, and many ignore all three operands.

NOTE

Opcodes are internal VM operation names, and they are not a part of the SQLite interface specification. Consequently, their operational semantics may change from one release to another. The SQLite development team does not encourage SQLite users to write bytecode programs on their own.

Table 6-1 displays a typical bytecode program that is equivalent to `SELECT * FROM t1`. The table `t1` has two columns, namely `x` and `y`. The top line on the table is not a part of the program. Every other line is a bytecode instruction.

Table 6-1. A typical bytecode program

Address	Opcode	P1	P2	P3
0	Goto	0	11	
1	Integer	0	0	

Table 6-1. A typical bytecode program

Address	Opcode	P1	P2	P3
2	OpenRead	0	2	#t1
3	SetNumColumn	0	2	
4	Rewind	0	9	
5	Column	0	0	#x
6	Column	0	1	#y
7	Callback	2	0	
8	Next	0	5	
9	Close	0	0	
10	Halt	0	0	
11	Transaction	0	0	
12	VerifyCookie	0	1	
13	Goto	0	1	

The VM is an interpreter, and here is its structure:

```
for (; pc < nOp && rc == SQLITE_OK; pc++){
   switch (aOp[pc].opcode){
   case OP_Add:
      /* Implementation of the ADD operation here */
      break;
   case OP_Goto:
      pc = op[pc].p2-1;
      break;
   case OP_Halt:
      pc = nOp;
      break;
   /* other cases for other opcodes */
   }
}
```

The interpreter is a simple loop containing a massive switch statement. Each case statement implements one bytecode instruction. (Opcode names are prefixed by `OP_`.) In each iteration, the VM fetches the next bytecode instruction from the program, i.e., from `aOp` array using `pc` (both are members of `Vdbe` object) as index into the array. It decodes and carries out the operation specified by the instruction. The VM begins an execution of a bytecode program starting at the instruction number 0.

The VM accesses a database using cursors. It can have zero or more open cursors on the database. Each cursor is a pointer into a single table or index tree. The cursor can seek to an entry with a particular key, or loop over all entries of the tree. The VM inserts new entries, retrieves the key/data at the current entry on the cursor, or deletes the entry.

The VM uses an operand stack and an arbitrary amount of numbered memory locations to hold all intermediate results. Many of the opcodes use operands from the stack. Computation results are also stored on the stack. Each stack or memory location holds a single data value. The memory locations are typically used to hold the result of a scalar SELECT that is part of a larger expression.

The VM continues a bytecode program execution until it processes a halt instruction or encounters an error (in the interpreter program, the `rc` variable stores the status of instruction executions), or the program counter points past the last instruction. When the VM halts, it releases all allocated memory, and closes all cursors. If the execution has stopped due to an error, the VM terminates the transaction or subtransaction, and removes changes made by the (sub)transaction from the database.

Section 6.2. Record Format

The VM composes data values into records, and stores them in B/B$^+$-trees. Each record consists of a key and optional data. The VM is solely responsible for maintaining internal structures of keys and data. (Although the B$^+$-tree module may split a single record across tree (leaf or internal) and multiple overflow pages, the VM sees the record as a logically contiguous byte string.) The VM uses two different but similar record formats for table and index records.

There are two ways to format data/key records: fixed length and variable length. For fixed length format, the same amount of space is used for all records (of a table or index); the size of each individual field is known at table/index creation time. In variable-length formatting, the space for an individual field may vary from one record to another. SQLite uses a variant of variable length record formatting because this has several advantages. It results in smaller database files because there is no wasted padding. It also makes the system run faster, as there are fewer bytes to move between the main memory and disk. In addition, the use of variable-length records allows SQLite to employ manifest typing instead of static typing.

Each primitive data value, stored in a database (or manipulated by the VM), has a data type associated with it. This is called the *storage type* of the data value. The operations that we can apply on the data value depend on the data type. Most SQL databases use static typing: A data type is associated with each column in a table, and only values of that particular data type are allowed to be stored in that column. SQLite relaxes this restriction by using `manifest typing`. In manifest typing, the data type is a property of the value itself, not of the column or variable in which the value is stored. SQLite uses manifest typing (even though the SQL language specification calls for static typing), where the storage type information is stored as a part of data value. SQLite allows users to store any value of any data type into any column regardless of the declared SQL type of that column. (There is an exception to this rule: an `integer primary key` column stores only integers.)

Figure 6-1. Format of table record
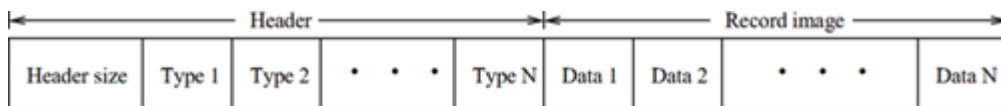
Format of table record (row data) is given in Figure 6-1. The record has two parts: header and record image. The header starts with a size field, followed by type fields. The header is followed by individual data items of the record. The header size is the number of bytes before the Data 1 field. The size is represented as a variable-length 64-bit integer in Huffman code, and it includes the number of bytes occupied by itself. The size effectively acts as a pointer to the Data 1 item. After the header size field comes the data type fields, one for each data value in the record in the sequence of their appearance in the record. Each Type *i* field is a variable-length unsigned integer (the max value is 2^{64}) that encodes the storage type for Data *i*.

The VM supports five storage types: signed integer number, signed floating point number, character string, BLOB, and SQL NULL. Every data value stored in file or in-memory must be one of these five types. Note that some values may have multiple representations at a time. For example, 123 can be an integer number, a floating point number, and a string, all at once. BLOB and NULL values cannot have another representation. An implicit conversion from one type to another occurs as necessary.

The integer value encodings of the types are given in Table 6-2. The beauty of this encoding is that the data length becomes a part of the type encoding.

Table 6-2. Storage types and their meanings

Type value	Meaning	Length of data
0	NULL	0
N in {1..4}	Signed integer	N

Table 6-2. Storage types and their meanings

Type value	Meaning	Length of data
5	Signed integer	6
6	Signed integer	8
7	IEEE float	8
8	Integer constant 0	0
9	Integer constant 1	0
10, 11	Reserved for expansion	N/A
N>=12 and even	Blob	(N-12)/2
N>=13 and odd	Text	(N-13)/2

The NULL type represents the SQL NULL value. For the INTEGER type, the data value is a signed integer number, stored in 1, 2, 3, 4, 6, or 8 bytes depending on the magnitude of the value. For the REAL type, the data value is a floating point number that is stored in 8 bytes, as specified in the IEEE floating point number representation standards. The type encoding values 8 and 9 represent integer constants 0 and 1, respectively. For the TEXT type, the data value is a text string, stored using the default encoding (UTF-8, UTF-16BE, or UTF-16-LE) format of the database file; for the latter two, the native byte ordering is either big-endian or little-endian, respectively. (Each database file stores text data in only one UTF format.) For the BLOB type, the data value is a blob, stored exactly as it was input.

In SQLite, every B$^+$-tree must have a unique key. Although a relation (table), by definition, does not contain identical rows, in practice, users may store duplicate rows in the relation. The database system must have means to treat them as different, distinct rows. The system must be capable of associating additional information for the differentiation purpose. It means that the system provides a new (unique) attribute for the relation. Thus, internally, each table has a unique primary key, and the key is either defined by the creator of the table or by SQLite.

For every SQL table, SQLite designates one column as the *rowid*(or `oid` or `_rowid_`) whose value uniquely identifies each row in the table. It is the implicit primary key for the table, the unique search key for the table B$^+$-tree. If any column of a table is declared as `integer primary key`, that column itself is treated (aliased) as the rowid for the table. Otherwise, SQLite creates a separate unique integer column for rowid, whose name is `rowid` itself. Thus, every table, with or without declaration of an `integer primary key` column, has a unique integer key, namely `rowid`. For this latter case, the rowid itself is internally treated as the integer primary key for the table. In either case, a rowid is a signed integer value in the range $-2^{63} \ldots 2^{63} - 1$.

Table 6-3 displays the contents of a typical SQL table, where the table is created by SQL statement: `create table t1(x ,y)`. The rowid column is added by SQLite. The rowid value is normally determined by SQLite. Nevertheless, you can insert any integer value for the rowid column, as in `insert into t1(rowid, x, y) values(100, 'hello', 'world')`.

Table 6-3. A typical SQL table with rowid as key, and x and y as data

rowid	x	y
-5	abc	xyz
1	abc	12345
2	456	def
100	hello	world
54321	NULL	987

NOTE

If rowid is an alias column (i.e., declared as INTEGER PRIMARY KEY), database users are responsible for the column values. If the rowid column is added by SQLite, then SQLite is responsible for generating the values, and it guarantees their uniqueness. When a row, without specifying the rowid column value, is inserted into the table, SQLite visits the table B$^+$-tree and finds an unused integer number for the rowid.

The rowid value, when stored as a part of data record, has an internal integer type. When stored as a key value, it is represented as a variable-length Huffman code. Negative rowids are allowed, but they always use nine bytes of storage, and so their use is discouraged. When rowids are generated by SQLite, they will always be non-negative, though you can explicitly give negative rowid values; the -5 in the previous table is an example.

In the previous section, you have seen that the key to each table's B$^+$-tree is an integer, and the data record is one row of the table. Indexes reverse this arrangement. For an index record, the key is the combined values of all indexing columns of the row being stored in the indexed table, and the data is an integer value that is the rowid of the row. To access a table row that has some particular values for indexing columns, SQLite first searches the index table to find the corresponding integer value, then uses that integer value to look up the complete record in the table's B$^+$-tree.

Figure 6-2. Format of index record

Header size	Type 1	Type 2	• • •	Type N	Data 1	Data 2	• • •	Data N	rowid

SQLite treats an index as a kind of table, and stores the index in its own B-tree. It maps a search key to a rowid. It can have its own key comparison, i.e., ordering function to order index entries. Each index record contains copies of indexing column value(s) followed by the rowid of the row being indexed. The format of index records is given in Figure 6-2. The entire record serves as the B-tree key; there is no data part. The encoding for index records is the same as that for the indexed table records, except that the rowid is appended, and the type of rowid does not appear in the record header because the type is always signed integer and is represented in Huffman code (and not in an internal integer type). (The other data values and their storage types are copied from the indexed table.) The contents of an index on column `x` is given in Table 6-4.

Table 6–4. An index on column x

x	rowid
NULL	54321
456	2
abc	-5
abc	1
hello	100

SQLite also supports multicolumn indexing. Table 6-5 presents the contents of an index with columns `y` and `x`. The entries in the index are ordered by their first column values; the second and subsequent columns are used as a tie-breaker.

Table 6–5. An index on columns y and x

y	x	rowid
987	NULL	54321
12345	abc	1
def	456	2
xyz	abc	-5
world	hello	100

Indexes are primarily used to speed up database search. For example, consider the following query: `SELECT y FROM t1 WHERE x=`. SQLite does an index search on index `t1(x)`, and finds all rowids for `x=`; for each rowid it searches the table `t1` B$^+$-tree to obtain the value for the `y` column for the row that satisifies the search.

Section 6.3. Data Type Management

SQLite data processing happens in the VM module. The VM is the sole manipulator of data stored in databases; everything else is controlled by bytecode programs that it executes. It decides what data to store there, and what data to retrieve from there. Assigning appropriate storage types to data values, and doing necessary type conversions, are the primary tasks of the VM. There are three places of data exchange where type conversions may take place: from application to engine, from engine to application, and from engine to engine. For the first two cases, the VM assigns types to user data. The VM attempts to convert user-supplied values into the declared SQL type of the column whenever it can, and vice versa. For the last item, the data conversions are required for expression evaluations. I discuss these three data conversion issues in the next three subsections.

In the Record Format section, I discussed storage formats for records of tables and indexes. Each record field value has a storage type. Every value supplied to SQLite, whether as literal embedded in an SQL statement, or as value bound to a prepared statement, is assigned a storage type before the statement is executed. The type is used to encode the supplied value to appropriate physical representation. The VM decides on the storage type for a given input value of a column in three steps: it first determines the storage type of the input data, then the declared SQL type of the column, and finally, if required, it does the type conversion. Under circumstances described next, SQLite may convert values between numeric storage types (INTEGER and REAL) and TEXT during query evaluation.

6.3.1.1. Input data type

The VM assigns initial storage types to user supplied values as follows. A value specified as literal as part of an SQL statement is assigned one of the following storage types:

- TEXT if the value is enclosed by single or double quotes
- INTEGER if the value is an unquoted number with no decimal point or exponent
- REAL if the value is an unquoted number with a decimal point or exponent
- NULL if the value is the character string NULL without any quote surrounding it
- BLOB if the value is specified using the X'ABCD' notation

Otherwise, the input value is rejected, and the query evaluation fails. Values for SQL parameters that are supplied using the `sqlite3_bind_*` API functions are assigned the storage types that most closely match the native type bound (e.g., `sqlite3_bind_blob` binds a value with storage type BLOB).

The storage type of a value that is the result of an SQL scalar operator depends on the outermost operator of the expression. User-defined functions may return values with any storage type. It is not generally possible to determine the type of the result of an expression at

SQL statement preparation time. The VM assigns the storage type at runtime on obtaining the value.

6.3.1.2. Column affinity

Even though each column (except integer primary key) can store any type of value, the value may have an affinity of its declared SQL type. Other SQL database engines that I am aware of use the more restrictive system of static typing, where the type is associated with the container, and not with the value. To maximize compatibility between SQLite and other database engines, SQLite supports the concept of *type affinity* on columns. The type affinity of a column is the recommended type for values stored in that column: "it is recommended, not required." Any column can still store any type of data. It is just that some columns, given the choice, will prefer to use one type over another.

NOTE

SQLite is typeless, i.e., there is no domain constraint. It permits storing any type of data in any table column irrespective of the declared SQL type of that column. (The exception to this rule is the rowid column; this column stores only integer values; any other value is rejected.) It lets you omit the SQL type declaration in `create table` statements. For example, `create table T1(a, b, c)` is a valid SQL statement in SQLite.

The preferred type for a column is called its affinity. Each column is assigned one of five affinity types: TEXT, NUMERIC, INTEGER, REAL, and NONE. (You may note a bit of naming conflict: "text," "integer," and "real" are names used for storage types, too. You can, however, determine the type category from the context.) The affinity type of a column is determined, depending on its declared SQL type in the CREATE TABLE statement, according to the following rules (SQLite is not at all strict about spelling errors in SQL type declarations):

1. If the SQL type contains the substring INT, then that column has INTEGER affinity.
2. If the SQL type contains any of the substrings CHAR, CLOB, or TEXT, then that column has TEXT affinity. (The SQL type VARCHAR contains the string CHAR, and therefore has the TEXT affinity.)
3. If the SQL type contains the substring BLOB, or if no type is specified, then the column has NONE affinity.
4. If the SQL type contains any of the substrings REAL, FLOA, or DOUB, then the column has REAL affinity.
5. Otherwise, the affinity type is NUMERIC.

The VM evaluates the rules in the same order as given above. The pattern matching is case insensitive. For example, if the declared SQL type of a column is BLOBINT, the affinity is INTEGER, and not NONE. If an SQL table is created using a `create table table1 as select...` statement, then all columns have no SQL types, and they are given NONE affinity. The type of implicit rowid is always integer.

6.3.1.3. Type conversion

There is a relationship between affinity types and storage types. If a user-supplied value for a column does not satisfy the relationship, the value is either rejected or converted into the appropriate format. When a value is inserted into a column, the VM first assigns the most suitable storage type (see the "Input data type" section), and then makes an attempt to convert the initial storage type into the format of its affinity type (see the "Column affinity" section). It does so using the following rules:

1. A column with TEXT affinity stores all data that have NULL, TEXT, or BLOB storage types. If a numerical value (integer or real) is inserted into the column, the value is converted into text form, and the final storage type becomes TEXT.
2. A column with NUMERIC affinity may contain values with all five storage types. When a text value is inserted into a NUMERIC column, an attempt is made to convert the value to an integer or real number. If the conversion is successful, then the converted value is stored using the INTEGER or REAL storage type. If the conversion cannot be performed, the value is stored using the TEXT storage type. No attempt is made to convert NULL or BLOB values.
3. A column with INTEGER affinity behaves in the same way as a column with NUMERIC affinity, except that if a real value with no floating point component (or text value that converts to such) is inserted, the value is converted to an integer and stored using the INTEGER storage type.
4. A column with REAL affinity behaves like a column with NUMERIC affinity, except that it forces integer values into floating point representation. (As an optimization, integer values are stored on disk as integers to take up less space, and are only converted to floating point as the values are read out of the table.)
5. A column with affinity NONE does not prefer one storage type over another. The VM does not attempt to convert any input value.

NOTE

All SQL database engines do data conversions. They reject input values that cannot be converted into desired types. The difference is that SQLite may still store the values even if a format conversion is not possible. For example, if you have a table column declared as SQL type INTEGER, and try to insert a string (e.g., "123" or "abc"), the VM will look at the string and see if it looks like a number. If the string does look like a number (as in "123"), it is converted into a number (and into an integer if the number does not have a fractional part), and is stored with real or integer storage type. But, if the string is not a well-formed number (as in the case "abc"), it is stored as a string with TEXT storage type. A column with TEXT affinity tries to convert numbers into an ASCII text representation before storing them. But BLOBs are stored in TEXT columns as BLOBs because SQLite cannot in general convert a BLOB into text. SQLite allows inserting string values into integer columns. This is a feature, not a bug.

6.3.1.4. A simple example
Figure 6-3. A typical table data record

CREATE TABLE T1(a, b, c);
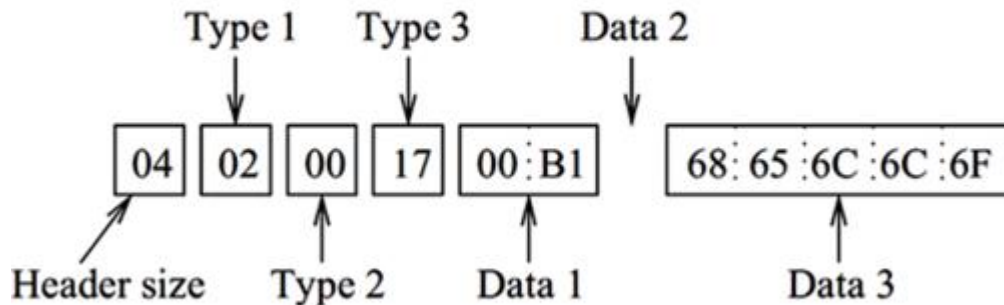INSERT INTO T1 VALUES(177, NULL, 'hello');

Table data record for the INSERT operation

Let us study a very simple example for more clarity. An example of a typical row record of a typical typeless table is given in Figure 6-3. The initial storage types for the a, b, and c column input values are integer, NULL, and text, respectively. The affinity type of all columns is NONE, and the VM does not make any attempt to convert the initial storage types. In the figure, the row record (header plus data items) consists of 11 bytes as follows. (All numbers in the record are given in hexadecimals.)

1. The header is 4 bytes long: one byte for the header size itself, plus one byte for each of three manifest types. The value 4 is encoded as the single byte 0x04.
2. Type 1 is the number 2 (representing a 2-byte signed integer) encoded as a single byte 0x02.
3. Type 2 is the number 0 (NULL) encoded as a single byte 0x00.
4. Type 3 is the number 23 (a text, (23 − 13)/2 = 5 bytes long) encoded as a single byte 0x17.
5. Data 1 is a 2-byte integer 00B1, which is the value 177. Note that 177 could not be encoded as a single byte because B1 is -79, not 177.
6. Data 2 is NULL, and it does not occupy any byte in the record.
7. Data 3 is the 5-byte string 68 65 6C 6C 6F. The zero-terminator is omitted.

The `sqlite3_column_*` API functions read data out of the SQLite engine. These functions attempt to convert the data value where appropriate. For example, if the internal representation is FLOAT, and a text result is requested (`sqlite3_column_text` function), the VM uses the `sprintf()` library function internally to do the conversion before returning the value to the caller. Table 6-6 presents data conversion rules that the VM applies on internal data to produce output data for applications.

Table 6-6. Data conversion protocol

Internal Type	Requested Type	Conversion
NULL	INTEGER	Result is 0
NULL	FLOAT	Result is 0.0
NULL	TEXT	Result is NULL pointer
NULL	BLOB	Result is NULL pointer
INTEGER	FLOAT	Convert from integer to float
INTEGER	TEXT	ASCII rendering of the integer
INTEGER	BLOB	Same as for INTEGER→TEXT
FLOAT	INTEGER	Convert from float to integer
FLOAT	TEXT	ASCII rendering of the float
FLOAT	BLOB	Same as FLOAT→TEXT
TEXT	INTEGER	Use `atoi() C library function`
TEXT	FLOAT	Use `atof() C library function`
TEXT	BLOB	No change
BLOB	INTEGER	Convert to TEXT then use `atoi()`
BLOB	FLOAT	Convert to TEXT then use `atof()`
BLOB	TEXT	Add a `\000` terminator if needed

The VM may convert internal data before comparing them one against another or evaluating expressions. It uses the following conversions for internal data.

6.3.3.1. Handling NULL values

The NULL value can be used for any table columns except primary key columns. The storage type of the NULL value is "NULL." The NULL value is distinct from all valid values for a given column irrespective of their storage types. SQL standards are not very specific about how to handle NULL values from columns in expressions. It is not clear from the standards exactly how NULL values should be handled in all circumstances. For example, how do we compare NULLs with one another, and with other values? SQLite handles NULLs in the way many other RDBMSs do. NULLs are indistinct (i.e., the same value) for the purpose of the SELECT DISTINCT statement, the UNION operator in a compound SELECT, and GROUP BY. NULLs are, however,

distinct in a UNIQUE column. NULLs are handled by the built-in SUM function as specified by the SQL standard. Arithmetic operations on NULL produce NULL.

6.3.3.2. Types for expressions

SQLite supports three kinds of comparison operations:

- binary comparison operators =, <, <=, >, >= and ! =
- set membership operator IN
- ternary comparison operator BETWEEN

The outcome of a comparison depends on the storage types of the two values being compared, according to the following rules:

1. A value (the lefthand side operand) with storage type NULL is considered less than any other value (including another value with storage type NULL).
2. An INTEGER or REAL value is less than any TEXT or BLOB value. When an INTEGER or REAL is compared to another INTEGER or REAL, a numerical comparison is performed.
3. A TEXT value is less than a BLOB value. When two TEXT values are compared, the standard C library function `memcmp` is usually used to determine the result. However this can be overridden by user-defined collation functions.
4. When two BLOB values are compared, the result is always determined using `memcmp`.

Before applying these rules, the first and foremost task for the VM is to determine the storage types of the operands of the comparison operator. It first determines the preliminary storage types of operands, and then, (if necessary) converts values between types based on their affinity. Finally, it performs the comparison using the above four rules.

If an expression is a column or a reference to a column via an AS alias or a subselect with a column as the return value or rowid, then the affinity of that column is used as the affinity of the expression. Otherwise, the expression does not have an SQL type, and its affinity is NONE. SQLite may attempt to convert values between the numeric storage types (INTEGER and REAL) and TEXT before performing a comparison operation. For binary comparisons, this is done in the cases enumerated next. The term "expression" in the bullets means any SQL scalar expression or literal other than a column value.

- When two column values are compared, if any one column has NUMERIC affinity, then that affinity is preferred for both values. That is, the VM makes an attempt to convert the other column value before the comparison.
- When a column value is compared to the result of an expression, the affinity of the column is applied to the result of the expression before the comparison takes place.

- When values of two expressions are compared, then no conversions occur. The values are compared following the above-mentioned four standard rules. For example, if a string is compared to a number, the number will always be less than the string.

In SQLite, the expression `a BETWEEN b AND c` is equivalent to `a >= b AND a <=`, even if this means that different affinities are applied to `a` in each of the two comparisons required to evaluate the expression.

Expressions of the kind `a IN (SELECT b ...)` are handled by the three rules enumerated above for = binary operator (e.g., `a = b`). For example if `b` is a column value, and `a` is an expression, then the affinity of `b` is applied to `a` before any comparisons take place. SQLite treats the expression `a IN (x, y, z)` as equivalent to `a = z OR a = y OR a = z` even if different affinities are applied to `a` in different equality expressions.

Here are a few simple examples. Suppose you have a table `t1` that is created by CREATE TABLE `t1(a TEXT, b NUMERIC, c BLOB)`. You insert a single row in the table by executing INSERT `INTO t1 VALUES ('500', '500', '500')`. The final storage types for `a`, `b`, and `c` column values are TEXT, INTEGER, TEXT, respectively.

- `SELECT a < 60, a < 40 FROM t1` converts 60 and 40 to "60" and "40," respectively, before the comparison, because the `a` column has TEXT affinity and the values are compared as TEXT; and the statement returns `1|0` as output because "500" is less than "60" as text, and is not less than "40."
- `SELECT b < 60, b < 600 FROM t1` does not convert any literal values, and compares values as NUMERIC, and returns `0|1` as output.
- `SELECT c < 60, c < 600 from t1` does not convert 60 and 600 because `c` has NONE affinity. The two values (storage class NUMERIC) are less than "500" (storage class TEXT), and the SELECT returns `0|0` as output.

6.3.3.3. Operator types

All mathematical operators (except the concatenation operator ||) apply NUMERIC affinity to all operands prior to their evaluation. If any operand cannot be converted to NUMERIC, then the result of the operation is NULL. For the concatenation operator, TEXT affinity is applied to both operands. If either operand cannot be converted to TEXT (because it is NULL or BLOB), then the result of the concatenation is NULL.

6.3.3.4. Types in ORDER BY

When values are sorted by an ORDER BY clause, no storage type conversions take place before the sort. The previously stated standard comparison rules are followed: values with storage type NULL come first, followed by INTEGER and REAL values interspersed in numeric order, followed by TEXT values, usually in `memcmp()` order, and, finally, BLOB values in `memcmp()` order. Texts can be ordered by user-defined collation functions by overriding the `memcmp()`

function. (Collations are user-defined sorting functions. You can refer to the SQLite homepage to learn how to use collations in SQL statements.)

6.3.3.5. Types in GROUP BY

When grouping values with the GROUP BY clause, no storage type conversions take place before grouping. Values with different storage types are considered distinct, except for INTEGER and REAL values, which are considered equal if they are numerically equal.

6.3.3.6. Types in Compound SELECTs

The compound SELECT operators (namely UNION, INTERSECT, and EXCEPT), perform implicit comparisons between values. Before these comparisons are performed, an affinity may be applied to each value. The same affinity, if any, is applied to all values that may be returned in a single column of the compound SELECT result set. The affinity applied is the affinity of the column returned by the leftmost component SELECT that has a column value (and not some other kind of expression) in that position. If for a given compound SELECT column none of the component SELECTs return a column value, no affinity is applied to the values from that column before they are compared.

Chapter 7. Further Information

In previous sections I discussed design principles, engineering trade offs, and implementation issues of the SQLite engine, and presented some simple SQLite applications. Due to brevity of space, I did not discuss the SQLite frontend parsing system. I encourage you to visit the SQLite homepage at http://www.sqlite.org and follow the links there to get more information about the technology. You can also subscribe to the SQLite mailing list by sending an email to sqlite-users-subscribe@sqlite.org. The SQLite source code is well commented and an excellent source of information on SQLite internals. You can learn more about SQLite by actively using the technology.