The C language interface to the SQLite library

(This page was last modified on 2002/07/13 17:18:37 UTC)

The SQLite library is designed to be very easy to use from a C or C++ program. This document gives an overview of the C/C++ programming interface.

The Core API

The interface to the SQLite library consists of three core functions, one opaque data structure, and some constants used as return values. The core interface is as follows:

```
typedef struct sqlite sqlite;
#define SQLITE_OK           0   /* Successful result */

sqlite *sqlite_open(const char *dbname, int mode, char **errmsg);

void sqlite_close(sqlite*);

int sqlite_exec(
  sqlite*,
  char *sql,
  int (*)(void*,int,char**,char**),
  void*,
  char **errmsg
);
```

The above is all you really need to know in order to use SQLite in your C or C++ programs. There are other convenience functions available (and described below) but we will begin by describing the core functions shown above.

Opening a database

Use the **sqlite_open()** function to open an existing SQLite database or to create a new SQLite database. The first argument is the database name. The second argument is intended to signal whether the database is going to be used for reading and writing or just for reading. But in the current implementation, the second argument to **sqlite_open** is ignored. The third argument is a pointer to a string pointer. If the third argument is not NULL and an error occurs while trying to open the database, then an error message will be written to memory obtained from malloc() and *errmsg will be made to point to this error message. The calling function is responsible for freeing the memory when it has finished with it.

The name of an SQLite database is the name of a file that will contain the database. If the file does not exist, SQLite attempts to create and initialize it. If the file is read-only (due to permission bits or because it is located on read-only media like a CD-ROM) then SQLite opens the database for reading only. The entire SQL database is stored in a single file on the disk. But additional temporary files may be created during the execution of an SQL command in order to store the database rollback journal or temporary and intermediate results of a query.

The return value of the **sqlite_open()** function is a pointer to an opaque **sqlite** structure. This pointer will be the first argument to all subsequent SQLite function calls that deal with the same database.

NULL is returned if the open fails for any reason.

Closing the database

To close an SQLite database, call the **sqlite_close()** function passing it the sqlite structure pointer that was obtained from a prior call to **sqlite_open**. If a transaction is active when the database is closed, the transaction is rolled back.

Executing SQL statements

The **sqlite_exec()** function is used to process SQL statements and queries. This function requires 5 parameters as follows:

1. A pointer to the sqlite structure obtained from a prior call to **sqlite_open()**.

2. A null-terminated string containing the text of one or more SQL statements and/or queries to be processed.

3. A pointer to a callback function which is invoked once for each row in the result of a query. This argument may be NULL, in which case no callbacks will ever be invoked.

4. A pointer that is forwarded to become the first argument to the callback function.

5. A pointer to an error string. Error messages are written to space obtained from malloc() and the error string is made to point to the malloced space. The calling function is responsible for freeing this space when it has finished with it. This argument may be NULL, in which case error messages are not reported back to the calling function.

The callback function is used to receive the results of a query. A prototype for the callback function is as follows:

```
int Callback(void *pArg, int argc, char **argv, char **columnNames){
  return 0;
}
```

The first argument to the callback is just a copy of the fourth argument to **sqlite_exec()** This parameter can be used to pass arbitrary information through to the callback function from client code. The second argument is the number of columns in the query result. The third argument is an array of pointers to strings where each string is a single column of the result for that record. Note that the callback function reports a NULL value in the database as a NULL pointer, which is very different from an empty string. If the i-th parameter is an empty string, we will get:

```
argv[i][0] == 0
```

But if the i-th parameter is NULL we will get:

```
argv[i] == 0
```

The names of the columns are contained in the fourth argument.

If the EMPTY_RESULT_CALLBACKS pragma is set to ON and the result of a query is an empty set, then the callback is invoked once with the third parameter (argv) set to 0. In other words

```
argv == 0
```

The second parameter (argc) and the fourth parameter (columnNames) are still valid and can be used to determine the number and names of the result columns if there had been a result. The default behavior is not to invoke the callback at all if the result set is empty.

The callback function should normally return 0. If the callback function returns non-zero, the query is immediately aborted and **sqlite_exec()** will return SQLITE_ABORT.

Error Codes

The **sqlite_exec()** function normally returns SQLITE_OK. But if something goes wrong it can return a different value to indicate the type of error. Here is a complete list of the return codes:

```
#define SQLITE_OK           0   /* Successful result */
#define SQLITE_ERROR        1   /* SQL error or missing database */
#define SQLITE_INTERNAL     2   /* An internal logic error in SQLite */
#define SQLITE_PERM         3   /* Access permission denied */
#define SQLITE_ABORT        4   /* Callback routine requested an abort */
#define SQLITE_BUSY         5   /* The database file is locked */
#define SQLITE_LOCKED       6   /* A table in the database is locked */
#define SQLITE_NOMEM        7   /* A malloc() failed */
#define SQLITE_READONLY     8   /* Attempt to write a readonly database */
#define SQLITE_INTERRUPT    9   /* Operation terminated by sqlite_interrupt() *
#define SQLITE_IOERR       10   /* Some kind of disk I/O error occurred */
#define SQLITE_CORRUPT     11   /* The database disk image is malformed */
#define SQLITE_NOTFOUND    12   /* (Internal Only) Table or record not found */
#define SQLITE_FULL        13   /* Insertion failed because database is full */
#define SQLITE_CANTOPEN    14   /* Unable to open the database file */
#define SQLITE_PROTOCOL    15   /* Database lock protocol error */
#define SQLITE_EMPTY       16   /* (Internal Only) Database table is empty */
#define SQLITE_SCHEMA      17   /* The database schema changed */
#define SQLITE_TOOBIG      18   /* Too much data for one row of a table */
#define SQLITE_CONSTRAINT  19   /* Abort due to contraint violation */
#define SQLITE_MISMATCH    20   /* Data type mismatch */
#define SQLITE_MISUSE      21   /* Library used incorrectly */
```

The meanings of these various return values are as follows:

SQLITE_OK

> This value is returned if everything worked and there were no errors.

SQLITE_INTERNAL

> This value indicates that an internal consistency check within the SQLite library failed. This can only happen if there is a bug in the SQLite library. If you ever get an SQLITE_INTERNAL reply from an **sqlite_exec()** call, please report the problem on the SQLite mailing list.

SQLITE_ERROR

This return value indicates that there was an error in the SQL that was passed into the **sqlite_exec()**.

SQLITE_PERM

This return value says that the access permissions on the database file are such that the file cannot be opened.

SQLITE_ABORT

This value is returned if the callback function returns non-zero.

SQLITE_BUSY

This return code indicates that another program or thread has the database locked. SQLite allows two or more threads to read the database at the same time, but only one thread can have the database open for writing at the same time. Locking in SQLite is on the entire database.

SQLITE_LOCKED

This return code is similar to SQLITE_BUSY in that it indicates that the database is locked. But the source of the lock is a recursive call to **sqlite_exec()**. This return can only occur if you attempt to invoke sqlite_exec() from within a callback routine of a query from a prior invocation of sqlite_exec(). Recursive calls to sqlite_exec() are allowed as long as they do not attempt to write the same table.

SQLITE_NOMEM

This value is returned if a call to **malloc()** fails.

SQLITE_READONLY

This return code indicates that an attempt was made to write to a database file that is opened for reading only.

SQLITE_INTERRUPT

This value is returned if a call to **sqlite_interrupt()** interrupts a database operation in progress.

SQLITE_IOERR

This value is returned if the operating system informs SQLite that it is unable to perform some disk I/O operation. This could mean that there is no more space left on the disk.

SQLITE_CORRUPT

This value is returned if SQLite detects that the database it is working on has become corrupted. Corruption might occur due to a rogue process writing to the database file or it might happen due to an perviously undetected logic error in of SQLite. This value is also returned if a disk I/O error occurs in such a way that SQLite is forced to leave the database file in a corrupted state. The latter should only happen due to a hardware or operating system malfunction.

SQLITE_FULL

This value is returned if an insertion failed because there is no space left on the disk, or the database is too big to hold any more information. The latter case should only occur for databases that are larger than 2GB in size.

SQLITE_CANTOPEN

This value is returned if the database file could not be opened for some reason.

SQLITE_PROTOCOL

This value is returned if some other process is messing with file locks and has violated the file locking protocol that SQLite uses on its rollback journal files.

SQLITE_SCHEMA

When the database first opened, SQLite reads the database schema into memory and uses that schema to parse new SQL statements. If another process changes the schema, the command currently being processed will abort because the virtual machine code generated assumed the old schema. This is the return code for such cases. Retrying the command usually will clear the problem.

SQLITE_TOOBIG

SQLite will not store more than about 1 megabyte of data in a single row of a single table. If you attempt to store more than 1 megabyte in a single row, this is the return code you get.

SQLITE_CONSTRAINT

This constant is returned if the SQL statement would have violated a database constraint.

SQLITE_MISMATCH

This error occurs when there is an attempt to insert non-integer data into a column labeled INTEGER PRIMARY KEY. For most columns, SQLite ignores the data type and allows any kind of data to be stored. But an INTEGER PRIMARY KEY column is only allowed to store integer data.

SQLITE_MISUSE

This error might occur if one or more of the SQLite API routines is used incorrectly. Examples of incorrect usage include calling **sqlite_exec()** after the database has been closed using **sqlite_close()** or calling **sqlite_exec()** with the same database pointer simultaneously from two separate threads.

The Extended API

Only the three core routines shown above are required to use SQLite. But there are many other functions that provide useful interfaces. These extended routines are as follows:

```
int sqlite_last_insert_rowid(sqlite*);

int sqlite_changes(sqlite*);

int sqlite_get_table(
  sqlite*,
  char *sql,
  char ***result,
  int *nrow,
  int *ncolumn,
  char **errmsg
);

void sqlite_free_table(char**);

void sqlite_interrupt(sqlite*);

int sqlite_complete(const char *sql);

void sqlite_busy_handler(sqlite*, int (*)(void*,const char*,int), void*);

void sqlite_busy_timeout(sqlite*, int ms);

const char sqlite_version[];

const char sqlite_encoding[];

int sqlite_exec_printf(
  sqlite*,
  char *sql,
  int (*)(void*,int,char**,char**),
  void*,
  char **errmsg,
  ...
);

int sqlite_exec_vprintf(
  sqlite*,
  char *sql,
  int (*)(void*,int,char**,char**),
  void*,
  char **errmsg,
  va_list
);

int sqlite_get_table_printf(
  sqlite*,
```

```
  char *sql,
  char ***result,
  int *nrow,
  int *ncolumn,
  char **errmsg,
  ...
);

int sqlite_get_table_vprintf(
  sqlite*,
  char *sql,
  char ***result,
  int *nrow,
  int *ncolumn,
  char **errmsg,
  va_list
);

char *sqlite_mprintf(const char *zFormat, ...);

char *sqlite_vmprintf(const char *zFormat, va_list);

void sqlite_freemem(char*);
```

All of the above definitions are included in the "sqlite.h" header file that comes in the source tree.

The ROWID of the most recent insert

Every row of an SQLite table has a unique integer key. If the table has a column labeled INTEGER PRIMARY KEY, then that column serves as the key. If there is no INTEGER PRIMARY KEY column then the key is a unique integer. The key for a row can be accessed in a SELECT statement or used in a WHERE or ORDER BY clause using any of the names "ROWID", "OID", or "_ROWID_".

When you do an insert into a table that does not have an INTEGER PRIMARY KEY column, or if the table does have an INTEGER PRIMARY KEY but the value for that column is not specified in the VALUES clause of the insert, then the key is automatically generated. You can find the value of the key for the most recent INSERT statement using the **sqlite_last_insert_rowid()** API function.

The number of rows that changed

The **sqlite_changes()** API function returns the number of rows that were inserted, deleted, or modified during the most recent **sqlite_exec()** call. The number reported includes any changes that were later undo by a ROLLBACK or ABORT. But rows that are deleted because of a DROP TABLE are *not* counted.

SQLite implements the command "**DELETE FROM table**" (without a WHERE clause) by dropping the table then recreating it. This is much faster than deleting the elements of the table individually. But it also means that the value returned from **sqlite_changes()** will be zero regardless of the number of elements that were originally in the table. If an accurate count of the number of elements deleted is necessary, use "**DELETE FROM table WHERE 1**" instead.

Querying without using a callback function

The **sqlite_get_table()** function is a wrapper around **sqlite_exec()** that collects all the information from successive callbacks and write it into memory obtained from malloc(). This is a convenience function that allows the application to get the entire result of a database query with a single function call.

The main result from **sqlite_get_table()** is an array of pointers to strings. There is one element in this array for each column of each row in the result. NULL results are represented by a NULL pointer. In addition to the regular data, there is an added row at the beginning of the array that contains the names of each column of the result.

As an example, consider the following query:

```
SELECT employee_name, login, host FROM users WHERE logic LIKE 'd%';
```

This query will return the name, login and host computer name for every employee whose login begins with the letter "d". If this query is submitted to **sqlite_get_table()** the result might look like this:

```
nrow = 2
ncolumn = 3
result[0] = "employee_name"
result[1] = "login"
result[2] = "host"
result[3] = "dummy"
result[4] = "No such user"
result[5] = 0
result[6] = "D. Richard Hipp"
result[7] = "drh"
result[8] = "zadok"
```

Notice that the "host" value for the "dummy" record is NULL so the result[] array contains a NULL pointer at that slot.

If the result set of a query is empty, then by default **sqlite_get_table()** will set nrow to 0 and leave its result parameter is set to NULL. But if the EMPTY_RESULT_CALLBACKS pragma is ON then the result parameter is initialized to the names of the columns only. For example, consider this query which has an empty result set:

```
SELECT employee_name, login, host FROM users WHERE employee_name IS NULL;
```

The default behavior gives this results:

```
nrow = 0
ncolumn = 0
result = 0
```

But if the EMPTY_RESULT_CALLBACKS pragma is ON, then the following is returned:

```
nrow = 0
ncolumn = 3
```

result[0] = "employee_name"
result[1] = "login"
result[2] = "host"

Memory to hold the information returned by **sqlite_get_table()** is obtained from malloc(). But the calling function should not try to free this information directly. Instead, pass the complete table to **sqlite_free_table()** when the table is no longer needed. It is safe to call **sqlite_free_table()** with a NULL pointer such as would be returned if the result set is empty.

The **sqlite_get_table()** routine returns the same integer result code as **sqlite_exec()**.

Interrupting an SQLite operation

The **sqlite_interrupt()** function can be called from a different thread or from a signal handler to cause the current database operation to exit at its first opportunity. When this happens, the **sqlite_exec()** routine (or the equivalent) that started the database operation will return SQLITE_INTERRUPT.

Testing for a complete SQL statement

The next interface routine to SQLite is a convenience function used to test whether or not a string forms a complete SQL statement. If the **sqlite_complete()** function returns true when its input is a string, then the argument forms a complete SQL statement. There are no guarantees that the syntax of that statement is correct, but we at least know the statement is complete. If **sqlite_complete()** returns false, then more text is required to complete the SQL statement.

For the purpose of the **sqlite_complete()** function, an SQL statement is complete if it ends in a semicolon.

The **sqlite** command-line utility uses the **sqlite_complete()** function to know when it needs to call **sqlite_exec()**. After each line of input is received, **sqlite** calls **sqlite_complete()** on all input in its buffer. If **sqlite_complete()** returns true, then **sqlite_exec()** is called and the input buffer is reset. If **sqlite_complete()** returns false, then the prompt is changed to the continuation prompt and another line of text is read and added to the input buffer.

Library version string

The SQLite library exports the string constant named **sqlite_version** which contains the version number of the library. The header file contains a macro SQLITE_VERSION with the same information. If desired, a program can compare the SQLITE_VERSION macro against the **sqlite_version** string constant to verify that the version number of the header file and the library match.

Library character encoding

By default, SQLite assumes that all data uses a fixed-size 8-bit character (iso8859). But if you give the --enable-utf8 option to the configure script, then the library assumes UTF-8 variable sized characters. This makes a difference for the LIKE and GLOB operators and the LENGTH() and SUBSTR() functions. The static string **sqlite_encoding** will be set to either "UTF-8" or "iso8859" to indicate how the library was compiled. In addition, the **sqlite.h** header file will define one of the macros **SQLITE_UTF8** or

SQLITE_ISO8859, as appropriate.

Note that the character encoding mechanism used by SQLite cannot be changed at run-time. This is a compile-time option only. The **sqlite_encoding** character string just tells you how the library was compiled.

Changing the library's response to locked files

The **sqlite_busy_handler()** procedure can be used to register a busy callback with an open SQLite database. The busy callback will be invoked whenever SQLite tries to access a database that is locked. The callback will typically do some other useful work, or perhaps sleep, in order to give the lock a chance to clear. If the callback returns non-zero, then SQLite tries again to access the database and the cycle repeats. If the callback returns zero, then SQLite aborts the current operation and returns SQLITE_BUSY.

The arguments to **sqlite_busy_handler()** are the opaque structure returned from **sqlite_open()**, a pointer to the busy callback function, and a generic pointer that will be passed as the first argument to the busy callback. When SQLite invokes the busy callback, it sends it three arguments: the generic pointer that was passed in as the third argument to **sqlite_busy_handler**, the name of the database table or index that the library is trying to access, and the number of times that the library has attempted to access the database table or index.

For the common case where we want the busy callback to sleep, the SQLite library provides a convenience routine **sqlite_busy_timeout()**. The first argument to **sqlite_busy_timeout()** is a pointer to an open SQLite database and the second argument is a number of milliseconds. After **sqlite_busy_timeout()** has been executed, the SQLite library will wait for the lock to clear for at least the number of milliseconds specified before it returns SQLITE_BUSY. Specifying zero milliseconds for the timeout restores the default behavior.

Using the `_printf()` wrapper functions

The four utility functions

- **sqlite_exec_printf()**
- **sqlite_exec_vprintf()**
- **sqlite_get_table_printf()**
- **sqlite_get_table_vprintf()**

implement the same query functionality as **sqlite_exec()** and **sqlite_get_table()**. But instead of taking a complete SQL statement as their second argument, the four **_printf** routines take a printf-style format string. The SQL statement to be executed is generated from this format string and from whatever additional arguments are attached to the end of the function call.

There are two advantages to using the SQLite printf functions instead of **sprintf()**. First of all, with the SQLite printf routines, there is never a danger of overflowing a static buffer as there is with **sprintf()**. The SQLite printf routines automatically allocate (and later free) as much memory as is necessary to hold the SQL statements generated.

The second advantage the SQLite printf routines have over **sprintf()** are two new formatting options

specifically designed to support string literals in SQL. Within the format string, the %q formatting option works very much like %s in that it reads a null-terminated string from the argument list and inserts it into the result. But %q translates the inserted string by making two copies of every single-quote (') character in the substituted string. This has the effect of escaping the end-of-string meaning of single-quote within a string literal. The %Q formatting option works similar; it translates the single-quotes like %q and additionally encloses the resulting string in single-quotes. If the argument for the %Q formatting options is a NULL pointer, the resulting string is NULL without single quotes.

Consider an example. Suppose you are trying to insert a string value into a database table where the string value was obtained from user input. Suppose the string to be inserted is stored in a variable named zString. The code to do the insertion might look like this:

```
sqlite_exec_printf(db,
  "INSERT INTO table1 VALUES('%s')",
  0, 0, 0, zString);
```

If the zString variable holds text like "Hello", then this statement will work just fine. But suppose the user enters a string like "Hi y'all!". The SQL statement generated reads as follows:

```
INSERT INTO table1 VALUES('Hi y'all')
```

This is not valid SQL because of the apostrophy in the word "y'all". But if the %q formatting option is used instead of %s, like this:

```
sqlite_exec_printf(db,
  "INSERT INTO table1 VALUES('%q')",
  0, 0, 0, zString);
```

Then the generated SQL will look like the following:

```
INSERT INTO table1 VALUES('Hi y''all')
```

Here the apostrophy has been escaped and the SQL statement is well-formed. When generating SQL on-the-fly from data that might contain a single-quote character ('), it is always a good idea to use the SQLite printf routines and the %q formatting option instead of **sprintf**.

If the %Q formatting option is used instead of %q, like this:

```
sqlite_exec_printf(db,
  "INSERT INTO table1 VALUES(%Q)",
  0, 0, 0, zString);
```

Then the generated SQL will look like the following:

```
INSERT INTO table1 VALUES('Hi y''all')
```

If the value of the zString variable is NULL, the generated SQL will look like the following:

```
INSERT INTO table1 VALUES(NULL)
```

All of the _printf() routines above are built around the following two functions:

```
char *sqlite_mprintf(const char *zFormat, ...);
char *sqlite_vmprintf(const char *zFormat, va_list);
```

The **sqlite_mprintf()** routine works like the the standard library **sprintf()** except that it writes its results into memory obtained from malloc() and returns a pointer to the malloced buffer. **sqlite_mprintf()** also understands the %q and %Q extensions described above. The **sqlite_vmprintf()** is a varargs version of the same routine. The string pointer that these routines return should be freed by passing it to **sqlite_freemem()**.

Adding New SQL Functions

Beginning with version 2.4.0, SQLite allows the SQL language to be extended with new functions implemented as C code. The following interface is used:

```
typedef struct sqlite_func sqlite_func;

int sqlite_create_function(
  sqlite *db,
  const char *zName,
  int nArg,
  void (*xFunc)(sqlite_func*,int,const char**),
  void *pUserData
);
int sqlite_create_aggregate(
  sqlite *db,
  const char *zName,
  int nArg,
  void (*xStep)(sqlite_func*,int,const char**),
  void (*xFinalize)(sqlite_func*),
  void *pUserData
);

char *sqlite_set_result_string(sqlite_func*,const char*,int);
void sqlite_set_result_int(sqlite_func*,int);
void sqlite_set_result_double(sqlite_func*,double);
void sqlite_set_result_error(sqlite_func*,const char*,int);

void *sqlite_user_data(sqlite_func*);
void *sqlite_aggregate_context(sqlite_func*, int nBytes);
int sqlite_aggregate_count(sqlite_func*);
```

The **sqlite_create_function()** interface is used to create regular functions and **sqlite_create_aggregate()** is used to create new aggregate functions. In both cases, the **db** parameter is an open SQLite database on which the functions should be registered, **zName** is the name of the new function, **nArg** is the number of arguments, and **pUserData** is a pointer which is passed through unchanged to the C implementation of the function.

For regular functions, the **xFunc** callback is invoked once for each function call. The implementation of xFunc should call one of the **sqlite_set_result_...** interfaces to return its result. The **sqlite_user_data()** routine can be used to retrieve the **pUserData** pointer that was passed in when the function was registered.

For aggregate functions, the **xStep** callback is invoked once for each row in the result and then **xFinalize** is invoked at the end to compute a final answer. The xStep routine can use the

sqlite_aggregate_context() interface to allocate memory that will be unique to that particular instance of the SQL function. This memory will be automatically deleted after xFinalize is called. The **sqlite_aggregate_count()** routine can be used to find out how many rows of data were passed to the aggregate. The xFinalize callback should invoke one of the **sqlite_set_result_...** interfaces to set the final result of the aggregate.

SQLite now implements all of its built-in functions using this interface. For additional information and examples on how to create new SQL functions, review the SQLite source code in the file **func.c**.

Usage Examples

For examples of how the SQLite C/C++ interface can be used, refer to the source code for the **sqlite** program in the file **src/shell.c** of the source tree. Additional information about sqlite is available at sqlite.html. See also the sources to the Tcl interface for SQLite in the source file **src/tclsqlite.c**.

 Back to the SQLite Home Page