APPENDIX A

SQL Reference

The SQLite library understands most of the standard SQL language. But it does omit some features while at the same time adding a few features of its own. This reference is an overview of the SQL syntax implemented by SQLite, taken directly from the original documentation on the SQLite website (`www.sqlite.org/lang.html`) and slightly edited. Many low-level productions are omitted. For detailed information on the language that SQLite understands, refer to the source code and the grammar file `parse.y` in the source distribution.

In all of the syntax diagrams that follow, literal text is shown in bold. Nonterminal symbols are shown in italic. Operators that are part of the syntactic markup itself are shown as regular, unformatted text.

ALTER TABLE

sql-statement ::= **ALTER TABLE** [*database-name* **.**] *table-name alteration*
alteration ::= **RENAME TO** *new-table-name*
alteration ::= **ADD** [**COLUMN**] *column-def*

SQLite's version of the `ALTER TABLE` command allows the user to rename or add a new column to an existing table. It is not possible to remove a column from a table.

ANALYZE

sql-statement ::= **ANALYZE**
sql-statement ::= **ANALYZE** *database-name*
sql-statement ::= **ANALYZE** [*database-name* **.**] *table-name*

The `ANALYZE` command gathers statistics about indexes and stores them in a special table in the database where the query optimizer can use them to help make better index choices. If no arguments are given, all indexes in all attached databases are analyzed. If a database name is given as the argument, all indexes in that one database are analyzed. If the argument is a table name, then only indexes associated with that one table are analyzed.

The initial implementation stores all statistics in a single table named `sqlite_stat1`. Future enhancements may create additional tables with the same name pattern except with the "1" changed to a different digit. The `sqlite_stat1` table cannot be dropped, but all the content can be deleted, which has the same effect.

The RENAME TO syntax is used to rename the table identified by database-name.table-name to new-table-name. This command cannot be used to move a table between attached databases, but only to rename a table within the same database.

If the table being renamed has triggers or indexes, then these remain attached to the table after it has been renamed. However, if there are any view definitions, or statements executed by triggers that refer to the table being renamed, these are not automatically modified to use the new table name. If this is required, the triggers or view definitions must be dropped and re-created to use the new table name by hand.

The ADD COLUMN syntax is used to add a new column to an existing table. The new column is always appended to the end of the list of existing columns. column-def may take any of the forms permissible in a CREATE TABLE statement, with the following restrictions:

- The column may not have a PRIMARY KEY or UNIQUE constraint.
- The column may not have a default value of CURRENT_TIME, CURRENT_DATE, or CURRENT_TIMESTAMP.
- If a NOT NULL constraint is specified, then the column must have a default value other than NULL.

The execution time of the ALTER TABLE command is independent of the amount of data in the table. The ALTER TABLE command runs as quickly on a table with 10 million rows as it does on a table with 1 row.

After ADD COLUMN has been run on a database, that database will not be readable by SQLite version 3.1.3 and earlier until the database is vacuumed (see VACUUM).

ATTACH DATABASE

sql-statement ::= **ATTACH** [**DATABASE**] *database-filename* **AS** *database-name*

The ATTACH DATABASE statement adds a preexisting database file to the current database connection. If the filename contains punctuation characters it must be quoted. The names "main" and "temp" refer to the main database and the database used for temporary tables. These cannot be detached. Attached databases are removed using the DETACH DATABASE statement.

You can read from and write to an attached database and you can modify the schema of the attached database. This is a new feature of SQLite version 3.0. In SQLite 2.8, schema changes to attached databases were not allowed.

You cannot create a new table with the same name as a table in an attached database, but you can attach a database that contains tables whose names are duplicates of tables in the main database. It is also permissible to attach the same database file multiple times.

Tables in an attached database can be referred to using the syntax database-name.table-name. If an attached table doesn't have a duplicate table name in the main database, it doesn't require a database name prefix. When a database is attached, all of its tables that don't have duplicate names become the default table of that name. Any tables of that name attached afterward require the table prefix. If the default table of a given name is detached, then the last table of that name attached becomes the new default.

Transactions involving multiple attached databases are atomic, assuming that the main database is not :memory:. If the main database is :memory:, then transactions continue to be

atomic within each individual database file. But if the host computer crashes in the middle of a COMMIT where two or more database files are updated, some of those files might get the changes where others might not. Atomic commit of attached databases is a new feature of SQLite version 3.0. In SQLite version 2.8, all commits to attached databases behaved as if the main database were :memory:.

The maximum number of databases that can be attached to a given session is 10. This is a compile-time limit: the limit can be increased by altering the value in the source code and recompiling SQLite.

BEGIN TRANSACTION

sql-statement ::= **BEGIN** [**DEFERRED** | **IMMEDIATE** | **EXCLUSIVE**] [**TRANSACTION** [*name*]]
sql-statement ::= **END** [**TRANSACTION** [*name*]]
sql-statement ::= **COMMIT** [**TRANSACTION** [*name*]]
sql-statement ::= **ROLLBACK** [**TRANSACTION** [*name*]]

Beginning in version 2.0, SQLite supports transactions with rollback and atomic commit.

The optional transaction name is ignored. SQLite currently does not allow nested transactions.

No changes can be made to the database except within a transaction. Any command that changes the database (basically, any SQL command other than SELECT) will automatically start a transaction if one is not already in effect. Automatically started transactions are committed at the conclusion of the command.

Transactions can be started manually using the BEGIN command. Such transactions usually persist until the next COMMIT or ROLLBACK command. But a transaction will also ROLLBACK if the database is closed or if an error occurs and the ROLLBACK conflict resolution algorithm is specified. See the documentation on the ON CONFLICT clause for additional information about the ROLLBACK conflict resolution algorithm.

In SQLite version 3.0.8 and later, transactions can be deferred, immediate, or exclusive. Deferred means that no locks are acquired on the database until the database is first accessed. Thus with a deferred transaction, the BEGIN statement itself does nothing. Locks are not acquired until the first read or write operation. The first read operation against a database creates a SHARED lock and the first write operation creates a RESERVED lock. Because the acquisition of locks is deferred until they are needed, it is possible that another thread or process could create a separate transaction and write to the database after the BEGIN on the current thread has executed. If the transaction is immediate, then RESERVED locks are acquired on all databases as soon as the BEGIN command is executed, without waiting for the database to be used. After a BEGIN IMMEDIATE, you are guaranteed that no other thread or process will be able to write to the database or do a BEGIN IMMEDIATE or BEGIN EXCLUSIVE. Other processes can continue to read from the database, however. An exclusive transaction causes EXCLUSIVE locks to be acquired on all databases. After a BEGIN EXCLUSIVE, you are guaranteed that no other thread or process will be able to read or write the database until the transaction is complete.

SHARED, RESERVED, and EXCLUSIVE locks are described in detail in Chapters 4 and 5.

The default behavior for SQLite version 3.0.8 is a deferred transaction. For SQLite version 3.0.0 through 3.0.7, deferred is the only kind of transaction available. For SQLite version 2.8 and earlier, all transactions are exclusive.

The COMMIT command does not actually perform a commit until all pending SQL commands finish. Thus if two or more SELECT statements are in the middle of processing and a COMMIT is executed, the commit will not actually occur until all SELECT statements finish.

An attempt to execute COMMIT might result in an SQLITE_BUSY return code. This indicates that another thread or process had a read lock on the database that prevented the database from being updated. When COMMIT fails in this way, the transaction remains active and the COMMIT can be retried later after the reader has had a chance to clear.

comment

```
comment     ::= SQL-comment | C-comment
SQL-comment ::= -- single-line
C-comment   ::= /* multiple-lines [*/]
```

Comments aren't SQL commands, but can occur in SQL queries. They are treated as white space by the parser. They can begin anywhere white space can be found, including inside expressions that span multiple lines.

SQL comments only extend to the end of the current line.

C comments can span any number of lines. If there is no terminating delimiter, they extend to the end of the input. This is not treated as an error. A new SQL statement can begin on a line after a multiline comment ends. C comments can be embedded anywhere white space can occur, including inside expressions, and in the middle of other SQL statements. C comments do not nest. SQL comments inside a C comment will be ignored.

COMMIT TRANSACTION

See "BEGIN TRANSACTION."

COPY

```
sql-statement ::= COPY [ OR conflict-algorithm ] [database-name .]
                  table-name FROM filename [ USING DELIMITERS delim ]
```

The COPY command is available in SQLite version 2.8 and earlier. The COPY command has been removed from SQLite version 3.0 due to complications in trying to support it in a mixed UTF-8/16 environment. In version 3.0, the command-line shell contains a new command .import that can be used as a substitute for COPY.

The COPY command is an extension used to load large amounts of data into a table. It is modeled after a similar command found in PostgreSQL. In fact, the SQLite COPY command is specifically designed to be able to read the output of the PostgreSQL dump utility pg_dump so that data can be easily transferred from PostgreSQL into SQLite.

The table-name is the name of an existing table that is to be filled with data. The filename is a string or an identifier that names a file from which data will be read. The filename can be the STDIN to read data from standard input.

Each line of the input file is converted into a single record in the table. Columns are separated by tabs. If a tab occurs as data within a column, then that tab is preceded by a backslash (\) character. A backslash in the data appears as two backslashes in a row. The optional USING DELIMITERS clause can specify a delimiter other than a tab.

If a column consists of the characters \N, that column is filled with the value NULL.

The optional conflict-clause allows the specification of an alternative constraint conflict resolution algorithm to use this one command. See the section "ON CONFLICT" for additional information.

When the input data source is STDIN, the input can be terminated by a line that contains only a backslash and a dot (\.).

CREATE INDEX

```
sql-statement ::= CREATE [UNIQUE] INDEX [database-name .] index-name
                  ON table-name ( column-name [, column-name]* )
                  [ ON CONFLICT conflict-algorithm ]

column-name   ::= name [ COLLATE collation-name] [ ASC | DESC ]
```

The CREATE INDEX command consists of the keywords CREATE INDEX followed by the name of the new index, the keyword ON, the name of a previously created table that is to be indexed, and a parenthesized list of names of columns in the table that are used for the index key. Each column name can be followed by either the ASC or DESC keyword to indicate sort order, but the sort order is ignored in the current implementation. Sorting is always done in ascending order.

The COLLATE clause following each column name defines a collating sequence used for text entries in that column. The default collating sequence is the collating sequence defined for that column in the CREATE TABLE statement. Or if no collating sequence is otherwise defined, the built-in BINARY collating sequence is used.

There are no arbitrary limits on the number of indexes that can be attached to a single table, nor on the number of columns in an index.

If the UNIQUE keyword appears between CREATE and INDEX, then duplicate index entries are not allowed. Any attempt to insert a duplicate entry will result in an error.

The optional conflict-clause allows the specification of an alternative default constraint conflict resolution algorithm for this index. This only makes sense if the UNIQUE keyword is used since otherwise there are no constraints on the index. The default algorithm is ABORT. If a COPY, INSERT, or UPDATE statement specifies a particular conflict resolution algorithm, that algorithm is used in place of the default algorithm specified here. See the section "ON CONFLICT" for additional information.

The exact text of each CREATE INDEX statement is stored in the sqlite_master or sqlite_temp_master table, depending on whether the table being indexed is temporary. Every time the database is opened, all CREATE INDEX statements are read from the sqlite_master table and used to regenerate SQLite's internal representation of the index layout.

Indexes are removed with the DROP INDEX command.

CREATE TABLE

```
sql-command ::= CREATE [TEMP | TEMPORARY] TABLE table-name (
                column-def [, column-def]* [, constraint]* )

sql-command ::= CREATE [TEMP | TEMPORARY] TABLE [database-name.]
                table-name AS select-statement

column-def  ::= name [type] [[CONSTRAINT name] column-constraint]*

type        ::= typename | typename ( number ) | typename (number, number)

column-constraint ::= NOT NULL [ conflict-clause ] |
  PRIMARY KEY [sort-order] [ conflict-clause ] [AUTOINCREMENT] |
  UNIQUE [ conflict-clause ] |
  CHECK ( expr ) [ conflict-clause ] |
  DEFAULT value |
  COLLATE collation-name

constraint ::= PRIMARY KEY ( column-list ) [ conflict-clause ] |
  UNIQUE ( column-list ) [ conflict-clause ] |
  CHECK ( expr ) [ conflict-clause ]

conflict-clause ::= ON CONFLICT conflict-algorithm
```

A `CREATE TABLE` statement is basically the keywords `CREATE TABLE` followed by the name of a new table and a parenthesized list of column definitions and constraints. The table name can be either an identifier or a string. Table names that begin with `sqlite_` are reserved for use by SQLite.

Each column definition is the name of the column followed by the data type for that column, then one or more optional column constraints. The data type for the column does not restrict what data may be put in that column. The `UNIQUE` constraint causes an index to be created on the specified columns. This index must contain unique keys. The `COLLATE` clause specifies what text collating function to use when comparing text entries for the column. The built-in `BINARY` collating function is used by default.

The `DEFAULT` constraint specifies a default value to use when doing an `INSERT`. The value may be `NULL`, a string constant, or a number. Starting with version 3.1.0, the default value may also be one of the special case-independent keywords `CURRENT_TIME`, `CURRENT_DATE`, or `CURRENT_TIMESTAMP`. If the value is `NULL`, a string constant, or number, it is literally inserted into the column whenever an `INSERT` statement that does not specify a value for the column is executed. If the value is `CURRENT_TIME`, `CURRENT_DATE`, or `CURRENT_TIMESTAMP`, then the current UTC date and/or time is inserted into the columns. For `CURRENT_TIME`, the format is *HH*:*MM*:*SS*. For `CURRENT_DATE`, it is *YYYY-MM-DD*. The format for `CURRENT_TIMESTAMP` is *YYYY-MM-DD HH*:*MM*:*SS*.

Specifying a `PRIMARY KEY` normally just creates a `UNIQUE` index on the corresponding columns. However, if the primary key is on a single column that has data type `INTEGER`, then that column is used internally as the actual key of the B-tree for the table. This means that the column may only hold unique integer values. (Except in this one case, SQLite ignores the data type specification of columns and allows any kind of data to be put in a column regardless of its declared

data type.) If a table does not have an INTEGER PRIMARY KEY column, then the B-tree key will be an automatically generated integer. The B-tree key for a row can always be accessed using one of the special names ROWID, OID, or _ROWID_. This is true regardless of whether there is an INTEGER PRIMARY KEY. An INTEGER PRIMARY KEY column may also include the keyword AUTOINCREMENT. The AUTOINCREMENT keyword modified the way that B-tree keys are automatically generated. For additional information on AUTOINCREMENT, see the "Autoincrement Values" section later in this appendix.

If the TEMP or TEMPORARY keyword occurs in between CREATE and TABLE, then the table that is created is only visible to the process that opened the database and is automatically deleted when the database is closed. Any indexes created on a temporary table are also temporary. Temporary tables and indexes are stored in a separate file distinct from the main database file.

If a database-name is specified, then the table is created in the named database. It is an error to specify both a database-name and the TEMP keyword, unless the database-name is "temp". If no database name is specified, and the TEMP keyword is not present, the table is created in the main database.

The optional conflict-clause following each constraint allows the specification of an alternative default constraint conflict resolution algorithm for that constraint. The default is ABORT. Different constraints within the same table may have different default conflict resolution algorithms. If a COPY, INSERT, or UPDATE command specifies a different conflict resolution algorithm, then that algorithm is used in place of the default algorithm specified in the CREATE TABLE statement. See "ON CONFLICT" for additional information.

CHECK constraints are implemented in SQLite version 3.3.0 and later. As of version 2.3.0, only NOT NULL, PRIMARY KEY, and UNIQUE constraints are supported.

There are no arbitrary limits on the number of columns or on the number of constraints in a table. The total amount of data in a single row is limited to about 1MB in version 2.8. In version 3.0 there is no arbitrary limit on the amount of data in a row.

The CREATE TABLE AS form defines the table to be the result set of a query. The names of the table columns are the names of the columns in the result.

The exact text of each CREATE TABLE statement is stored in the sqlite_master table. Every time the database is opened, all CREATE TABLE statements are read from the sqlite_master table and used to regenerate SQLite's internal representation of the table layout. If the original command was a CREATE TABLE AS, then an equivalent CREATE TABLE statement is synthesized and stored in sqlite_master in place of the original command. The text of CREATE TEMPORARY TABLE statements is stored in the sqlite_temp_master table.

Tables are removed using the DROP TABLE statement.

Autoincrement Values

In SQLite, every row of every table has an integer ROWID. The ROWID for each row is unique among all rows in the same table. In SQLite version 2.8 the ROWID is a 32-bit signed integer. Version 3.0 of SQLite expanded the ROWID to be a 64-bit signed integer.

You can access the ROWID of a SQLite table using one of the special column names ROWID, _ROWID_, or OID. Unless you declare an ordinary table column to use one of those special names, the use of that name will refer to the declared column and not to the internal ROWID.

If a table contains a column of type INTEGER PRIMARY KEY, then that column becomes an alias for the ROWID. You can then access the ROWID using any of four different names, the original

three names described earlier or the name given to the INTEGER PRIMARY KEY column. All these names are aliases for one another and work equally well in any context.

When a new row is inserted into a SQLite table, the ROWID can either be specified as part of the INSERT statement or it can be assigned automatically by the database engine. To specify a ROWID manually, just include it in the list of values to be inserted. For example:

```
CREATE TABLE test1(a INT, b TEXT);
INSERT INTO test1(rowid, a, b) VALUES(123, 5, 'hello');
```

If no ROWID is specified on the insert, an appropriate ROWID is created automatically. The usual algorithm is to give the newly created row a ROWID that is 1 larger than the largest ROWID in the table prior to the insert. If the table is initially empty, then a ROWID of 1 is used. If the largest ROWID is equal to the largest possible integer (9223372036854775807 in SQLite version 3.0 and later), then the database engine starts picking candidate ROWIDs at random until it finds one that is not previously used.

The normal ROWID selection algorithm described here will generate monotonically increasing unique ROWIDs as long as you never use the maximum ROWID value and you never delete the entry in the table with the largest ROWID. If you ever delete rows or if you ever create a row with the maximum possible ROWID, then ROWIDs from previously deleted rows might be reused when creating new rows and newly created ROWIDs might not be in strictly ascending order.

The AUTOINCREMENT Keyword

If a column has the type INTEGER PRIMARY KEY AUTOINCREMENT, then a slightly different ROWID selection algorithm is used. The ROWID chosen for the new row is 1 larger than the largest ROWID that has ever before existed in that same table. If the table has never before contained any data, then a ROWID of 1 is used. If the table has previously held a row with the largest possible ROWID, then new INSERTs are not allowed and any attempt to insert a new row will fail with a SQLITE_FULL error.

SQLite keeps track of the largest ROWID that a table has ever held using the special sqlite_sequence table. The sqlite_sequence table is created and initialized automatically whenever a normal table that contains an AUTOINCREMENT column is created. The content of the sqlite_sequence table can be modified using ordinary UPDATE, INSERT, and DELETE statements. But making modifications to this table will likely perturb the AUTOINCREMENT key generation algorithm. Make sure you know what you are doing before you undertake such changes.

The behavior implemented by the AUTOINCREMENT keyword is subtly different from the default behavior. With AUTOINCREMENT, rows with automatically selected ROWIDs are guaranteed to have ROWIDs that have never been used before by the same table in the same database. And the automatically generated ROWIDs are guaranteed to be monotonically increasing. These are important properties in certain applications. But if your application does not need these properties, you should probably stay with the default behavior since the use of AUTOINCREMENT requires additional work to be done as each row is inserted and thus causes INSERTs to run a little slower.

CREATE TRIGGER

```
sql-statement ::= CREATE [TEMP | TEMPORARY] TRIGGER trigger-name
  [ BEFORE | AFTER ] database-event ON [database-name .] table-name
  trigger-action

sql-statement ::= CREATE [TEMP | TEMPORARY] TRIGGER trigger-name
  INSTEAD OF database-event ON [database-name .] view-name
  trigger-action

database-event ::= DELETE | INSERT | UPDATE | UPDATE OF column-list

trigger-action ::= [ FOR EACH ROW | FOR EACH STATEMENT ]
  [ WHEN expression ]
  BEGIN
      trigger-step ; [ trigger-step ; ]*
  END

trigger-step ::= update-statement | insert-statement |
  delete-statement | select-statement
```

The CREATE TRIGGER statement is used to add triggers to the database schema. Triggers are database operations (the trigger-action) that are automatically performed when a specified database event (the database-event) occurs.

A trigger may be specified to fire whenever a DELETE, INSERT, or UPDATE of a particular database table occurs, or whenever an UPDATE of one or more specified columns of a table are updated.

At this time SQLite supports only FOR EACH ROW triggers, not FOR EACH STATEMENT triggers. Hence explicitly specifying FOR EACH ROW is optional. FOR EACH ROW implies that the SQL statements specified as trigger-steps may be executed (depending on the WHEN clause) for each database row being inserted, updated, or deleted by the statement causing the trigger to fire.

Both the WHEN clause and the trigger-steps may access elements of the row being inserted, deleted, or updated using references of the form NEW.column-name and OLD.column-name, where column-name is the name of a column from the table that the trigger is associated with. OLD and NEW references may only be used in triggers on trigger-events for which they are relevant, as follows:

- INSERT: NEW references are valid
- UPDATE: NEW and OLD references are valid
- DELETE: OLD references are valid

If a WHEN clause is supplied, the SQL statements specified as trigger-steps are only executed for rows for which the WHEN clause is true. If no WHEN clause is supplied, the SQL statements are executed for all rows.

The specified trigger time determines when the trigger-steps will be executed relative to the insertion, modification, or removal of the associated row.

An ON CONFLICT clause may be specified as part of an UPDATE or INSERT trigger-step. However, if an ON CONFLICT clause is specified as part of the statement causing the trigger to fire, then this conflict-handling policy is used instead.

Triggers are automatically dropped when the table that they are associated with is dropped.

Triggers may be created on views, as well as ordinary tables, by specifying INSTEAD OF in the CREATE TRIGGER statement. If one or more ON INSERT, ON DELETE, or ON UPDATE triggers are defined on a view, then it is not an error to execute an INSERT, DELETE, or UPDATE statement on the view, respectively. Thereafter, executing an INSERT, DELETE, or UPDATE on the view causes the associated triggers to fire. The real tables underlying the view are not modified (except possibly explicitly, by a trigger program).

For example, assuming that customer records are stored in the customers table, and that order records are stored in the orders table, the following trigger ensures that all associated orders are redirected when a customer changes his or her address:

```
CREATE TRIGGER update_customer_address UPDATE OF address ON customers
BEGIN
  UPDATE orders SET address = new.address WHERE customer_name = old.name;
END;
```

With this trigger installed, executing the statement

```
UPDATE customers SET address = '1 Main St.' WHERE name = 'Jack Jones';
```

causes the following to be automatically executed:

```
UPDATE orders SET address = '1 Main St.' WHERE customer_name = 'Jack Jones';
```

Note that currently triggers may behave oddly when created on tables with INTEGER PRIMARY KEY fields. If a BEFORE trigger program modifies the INTEGER PRIMARY KEY field of a row that will be subsequently updated by the statement that causes the trigger to fire, then the update may not occur. The workaround is to declare the table with a PRIMARY KEY column instead of an INTEGER PRIMARY KEY column.

A special SQL function RAISE() may be used within a trigger program, with the following syntax:

```
raise-function ::= RAISE ( ABORT, error-message ) |
  RAISE ( FAIL, error-message ) |
  RAISE ( ROLLBACK, error-message ) |
  RAISE ( IGNORE )
```

When one of the first three forms is called during trigger program execution, the specified ON CONFLICT processing is performed (either ABORT, FAIL, or ROLLBACK) and the current query terminates. An error code of SQLITE_CONSTRAINT is returned to the user, along with the specified error message.

When RAISE(IGNORE) is called, the remainder of the current trigger program, the statement that caused the trigger program to execute, and any subsequent trigger programs that would have been executed are abandoned. No database changes are rolled back. If the statement that caused the trigger program to execute is itself part of a trigger program, then that trigger program resumes execution at the beginning of the next step.

Triggers are removed using the DROP TRIGGER statement.

CREATE VIEW

sql-command ::= **CREATE** [**TEMP** | **TEMPORARY**] **VIEW** [*database-name*.]
view-name **AS** *select-statement*

The CREATE VIEW command assigns a name to a prepackaged SELECT statement. Once the view is created, it can be used in the FROM clause of another SELECT in place of a table name.

If the TEMP or TEMPORARY keyword occurs in between CREATE and VIEW, then the view that is created is only visible to the process that opened the database and is automatically deleted when the database is closed.

If a database-name is specified, then the view is created in the named database. It is an error to specify both a database-name and the TEMP keyword, unless the database-name is "temp". If no database name is specified, and the TEMP keyword is not present, the table is created in the main database.

You cannot COPY, DELETE, INSERT, or UPDATE a view. Views are read-only in SQLite. However, in many cases you can use a TRIGGER on the view to accomplish the same thing. Views are removed with the DROP VIEW command.

DELETE

sql-statement ::= **DELETE FROM** [*database-name* .] *table-name* [**WHERE** *expr*]

The DELETE command is used to remove records from a table. The command consists of the DELETE FROM keywords followed by the name of the table from which records are to be removed. Without a WHERE clause, all rows of the table are removed. If a WHERE clause is supplied, then only those rows that match the expression are removed.

DETACH DATABASE

sql-command ::= **DETACH** [**DATABASE**] *database-name*

This statement detaches an additional database connection previously attached using the ATTACH DATABASE statement. It is possible to have the same database file attached multiple times using different names, and detaching one connection to a file will leave the others intact.

This statement will fail if SQLite is in the middle of a transaction.

DROP INDEX

sql-command ::= **DROP INDEX** [**IF EXISTS**] [*database-name* .] *index-name*

The DROP INDEX statement removes an index added with the CREATE INDEX statement. The index named is completely removed from the disk. The only way to recover the index is to reenter the appropriate CREATE INDEX command.

The DROP INDEX statement does not reduce the size of the database file in the default mode. Empty space in the database is retained for later INSERTs. To remove free space in the database, use the VACUUM command. If AUTOVACUUM mode is enabled for a database, then space will be freed automatically by DROP INDEX.

DROP TABLE

sql-command ::= **DROP TABLE** [**IF EXISTS**] [*database-name*.] *table-name*

The DROP TABLE statement removes a table added with the CREATE TABLE statement. The name specified is the table name. It is completely removed from the database schema and the disk file. The table cannot be recovered. All indexes associated with the table are also deleted.

The DROP TABLE statement does not reduce the size of the database file in the default mode. Empty space in the database is retained for later INSERTs. To remove free space in the database, use the VACUUM command. If AUTOVACUUM mode is enabled for a database, then space will be freed automatically by DROP TABLE.

The optional IF EXISTS clause suppresses the error that would normally result if the table does not exist.

DROP TRIGGER

sql-statement ::= **DROP TRIGGER** [*database-name* .] *trigger-name*

The DROP TRIGGER statement removes a trigger created by the CREATE TRIGGER statement. The trigger is deleted from the database schema. Note that triggers are automatically dropped when the associated table is dropped.

DROP VIEW

sql-command ::= **DROP VIEW** *view-name*

The DROP VIEW statement removes a view created by the CREATE VIEW statement. The name specified is the view name. It is removed from the database schema, but no actual data in the underlying base tables is modified.

END TRANSACTION

See “BEGIN TRANSACTION.”

EXPLAIN

sql-statement ::= **EXPLAIN** *sql-statement*

The EXPLAIN command modifier is a nonstandard extension. The idea comes from a similar command found in PostgreSQL, but the operation is completely different.

If the EXPLAIN keyword appears before any other SQLite SQL command, then instead of actually executing the command, the SQLite library will report back the sequence of virtual machine instructions it would have used to execute the command had the EXPLAIN keyword not been present. For additional information about virtual machine instructions, see Chapter 9.

expression

```
expr ::= expr binary-op expr |
         expr [NOT] like-op expr [ESCAPE expr] |
         unary-op expr |
         ( expr ) |
         column-name |
         table-name . column-name |
         database-name . table-name . column-name |
         literal-value |
         parameter |
         function-name ( expr-list | * ) |
         expr ISNULL |
         expr NOTNULL |
         expr [NOT] BETWEEN expr AND expr |
         expr [NOT] IN ( value-list ) |
         expr [NOT] IN ( select-statement ) |
         expr [NOT] IN [database-name .] table-name |
         [EXISTS] ( select-statement ) |
         CASE [expr] ( WHEN expr THEN expr )+ [ELSE expr] END |
         CAST ( expr AS type )

like-op ::= LIKE | GLOB | REGEXP
```

This section is different from the others. Whereas the other sections discuss a particular SQL command, this section does not focus on a standalone command but on "expressions," which are subcomponents of most other commands.

SQLite understands the following binary operators, in order from highest to lowest precedence:

```
||
*    /    %
+    -
<<   >>   &    |
<    <=   >    >=
=    ==   !=   <>   IN
AND
OR
```

Supported unary operators include the following:

```
-    +    !    ~
```

Note that there are two variations of the equals and non-equals operators. Equals can be either = or ==. The non-equals operator can be either != or <>. The || operator is "concatenate"—it joins together the two strings of its operands. The operator % outputs the remainder of its left operand modulo its right operand.

The result of any binary operator is a numeric value, except for the || concatenation operator, which gives a string result.

A literal value is an integer number or a floating-point number. Scientific notation is supported. The . character is always used as the decimal point even if the locale setting specifies, for this role—the use of , for the decimal point would result in syntactic ambiguity. A string constant is formed by enclosing the string in single quotes ('). A single quote within the string can be encoded by putting two single quotes in a row—as in Pascal. C-style escapes using the backslash character are not supported because they are not standard SQL. BLOB literals are string literals containing hexadecimal data and are preceded by a single x or X character. For example:

```
X'53514697465'
```

A literal value can also be the token NULL.

A parameter specifies a placeholder in the expression for a literal value that is filled in at runtime using the sqlite3_bind() API. Parameters can take several forms, as Table A-1 shows.

Table A-1. *Parameter Forms*

Format	Meaning
?NNN	A question mark followed by a number NNN holds a spot for the NNN-th parameter. NNN must be between 1 and 999.
?	A question mark that is not followed by a number holds a spot for the next unused parameter.
:AAAA	A colon followed by an identifier name holds a spot for a named parameter with the name AAAA. Named parameters are also numbered. The number assigned is the next unused number. To avoid confusion, it is best to avoid mixing named and numbered parameters.
$AAAA	A dollar sign followed by an identifier name also holds a spot for a named parameter with the name AAAA. The identifier name in this case can include one or more occurances of :: and a suffix enclosed in (...) containing any text at all. This syntax is the form of a variable name in the Tcl programming language.

Parameters that are not assigned values using sqlite3_bind() are treated as NULL.

The LIKE operator does a pattern matching comparison. The operand to the right contains the pattern, and the left-hand operand contains the string to match against the pattern. A percent symbol (%) in the pattern matches any sequence of zero or more characters in the string. An underscore in the pattern matches any single character in the string. Any other character matches itself or its lower/upper case equivalent (i.e., case-insensitive matching). (A bug: SQLite only understands upper/lowercase for 7-bit Latin characters. Hence the LIKE operator is case sensitive for 8-bit ISO8859 characters or UTF-8 characters. For example, the expression 'a' LIKE 'A' is true but 'æ' LIKE 'Æ' is false.)

If the optional ESCAPE clause is present, then the expression following the ESCAPE keyword must evaluate to a string consisting of a single character. This character may be used in the LIKE pattern to include literal percent or underscore characters. The escape character followed by a percent symbol, underscore, or itself matches a literal percent symbol, underscore, or escape character in the string, respectively.

The LIKE operator is not case sensitive and will match uppercase characters on one side against lowercase characters on the other.

The infix LIKE operator is implemented by calling the user function like(X,Y). If an ESCAPE clause is present, it adds a third parameter to the function call. The functionality of LIKE can be overridden by defining an alternative implementation of the like() SQL function.

The GLOB operator is similar to LIKE but uses the UNIX file globbing syntax for its wildcards. Also, GLOB is case sensitive, unlike LIKE. Both GLOB and LIKE may be preceded by the NOT keyword to invert the sense of the test. The infix GLOB operator is implemented by calling the user function glob(X,Y) and can be modified by overriding that function.

The REGEXP operator is a special syntax for the regexp() user function. No regexp() user function is defined by default and so use of the REGEXP operator will normally result in an error message. If a user-defined function named "regexp" is defined at runtime, that function will be called in order to implement the REGEXP operator.

A column name can be any of the names defined in the CREATE TABLE statement or one of the following special identifiers: ROWID, OID, or_ROWID_. These special identifiers all describe the unique random integer key (the "row key") associated with every row of every table. The special identifiers only refer to the row key if the CREATE TABLE statement does not define a real column with the same name. Row keys act like read-only columns. A row key can be used anywhere a regular column can be used, except that you cannot change the value of a row key in an UPDATE or INSERT statement. SELECT * ... does not return the row key.

SELECT statements can appear in expressions as either the right-hand operand of the IN operator, as a scalar quantity, or as the operand of an EXISTS operator. As a scalar quantity or the operand of an IN operator, the SELECT should have only a single column in its result. Compound SELECTs (connected with keywords like UNION or EXCEPT) are allowed. With the EXISTS operator, the columns in the result set of the SELECT are ignored and the expression returns true if one or more rows exist and false if the result set is empty. If no terms in the SELECT expression refer to value in the containing query, then the expression is evaluated once prior to any other processing and the result is reused as necessary. If the SELECT expression does contain variables from the outer query, then the SELECT is reevaluated every time it is needed.

When a SELECT is the right operand of the IN operator, the IN operator returns true if the result of the left operand is any of the values generated by the SELECT. The IN operator may be preceded by the NOT keyword to invert the sense of the test.

When a SELECT appears within an expression but is not the right operand of an IN operator, then the first row of the result of the SELECT becomes the value used in the expression. If the SELECT yields more than one result row, all rows after the first are ignored. If the SELECT yields no rows, then the value of the SELECT is NULL.

A CAST expression changes the data type of a column value into the type specified by type, which can be any non-empty type name that is valid for the type in a column definition of a CREATE TABLE statement.

Both simple and aggregate functions are supported. A simple function can be used in any expression. Simple functions return a result immediately based on their inputs. Aggregate functions may only be used in a SELECT statement. Aggregate functions compute their result across all rows of the result set.

The functions shown in Table A-2 are available by default. Additional functions may be written in C and added to the database engine using the sqlite3_create_function() API.

The aggregate functions shown in Table A-3 are available by default. Additional aggregate functions written in C may be added using the sqlite3_create_function() API.

Table A-2. *Built-in SQL Functions*

Function	Description
abs(X)	Return the absolute value of argument X.
coalesce(X,Y,...)	Return a copy of the first non-NULL argument. If all arguments are NULL then NULL is returned. There must be at least two arguments.
glob(X,Y)	This function is used to implement the X GLOB Y syntax of SQLite. The sqlite3_create_function() interface can be used to override this function and thereby change the operation of the GLOB operator.
ifnull(X,Y)	Return a copy of the first non-NULL argument. If both arguments are NULL then NULL is returned. This behaves the same as coalesce() above.
last_insert_rowid()	Return the ROWID of the last row insert from this connection to the database. This is the same value that would be returned from the sqlite_last_insert_rowid() API function.
length(X)	Return the string length of X in characters. If SQLite is configured to support UTF-8, then the number of UTF-8 characters is returned, not the number of bytes.
like(X,Y [,Z])	This function is used to implement the X LIKE Y [ESCAPE Z] syntax of SQL. If the optional ESCAPE clause is present, then the user function is invoked with three arguments. Otherwise, it is invoked with two arguments only. The sqlite_create_function() interface can be used to override this function and thereby change the operation of the LIKE operator. When doing this, it may be important to override both the two and three argument versions of the like() function. Otherwise, different code may be called to implement the LIKE operator depending on whether or not an ESCAPE clause was specified.
lower(X)	Return a copy of string X with all characters converted to lowercase. The C library tolower() routine is used for the conversion, which means that this function might not work correctly on UTF-8 characters.
max(X,Y,...)	Return the argument with the maximum value. Arguments may be strings in addition to numbers. The maximum value is determined by the usual sort order. Note that max() is a simple function when it has two or more arguments but converts to an aggregate function if given only a single argument.
min(X,Y,...)	Return the argument with the minimum value. Arguments may be strings in addition to numbers. The minimum value is determined by the usual sort order. Note that min() is a simple function when it has two or more arguments but converts to an aggregate function if given only a single argument.
nullif(X,Y)	Return the first argument if the arguments are different; otherwise return NULL.
quote(X)	This routine returns a string that is the value of its argument suitable for inclusion into another SQL statement. Strings are surrounded by single quotes with escapes on interior quotes as needed. BLOBs are encoded as hexadecimal literals. The current implementation of VACUUM uses this function. The function is also useful when writing triggers to implement undo/redo functionality.
random(*)	Return a random integer between -2147483648 and +2147483647.

Table A-2. *Built-in SQL Functions*

Function	Description
`round(X)`, `round(X,Y)`	Round off the number `X` to `Y` digits to the right of the decimal point. If the Y argument is omitted, 0 is assumed.
`soundex(X)`	Compute the soundex encoding of the string `X`. The string "?000" is returned if the argument is `NULL`. This function is omitted from SQLite by default. It is only available if the `-DSQLITE_SOUNDEX=1` compiler option is used when SQLite is built.
`sqlite_version(*)`	Return the version string for the SQLite library that is running. Example: "2.8.0"
`substr(X,Y,Z)`	Return a substring of input string `X` that begins with the `Y`-th character and that is `Z` characters long. The leftmost character of `X` is number 1. If `Y` is negative, then the first character of the substring is found by counting from the right rather than the left. If SQLite is configured to support UTF-8, then characters indexes refer to actual UTF-8 characters, not bytes.
`typeof(X)`	Return the type of the expression `X`. The only return values are "null", "integer", "real", "text", and "blob".
`upper(X)`	Return a copy of input string `X` converted to all uppercase letters. The implementation of this function uses the C library routine `toupper()`, which means it may not work correctly on UTF-8 strings.

In any aggregate function that takes a single argument, that argument can be preceded by the keyword `DISTINCT`. In such cases, duplicate elements are filtered before being passed into the aggregate function. For example, the function `count(distinct X)` will return the number of distinct values of column `X` instead of the total number of non-`NULL` values in column `X`.

Table A-3. *SQLite Built-in Aggregate Functions*

Function	Description
`avg(X)`	Return the average value of all non-`NULL` `X` within a group. String and `BLOB` values that do not look like numbers are interpreted as 0. The result of `avg()` is always a floating-point value even if all inputs are integers.
`count(X)`, `count(*)`	The first form returns a count of the number of times that `X` is not `NULL` in a group. The second form returns the total number of rows in the group.
`max(X)`	Return the maximum value of all values in the group. The usual sort order is used to determine the maximum.
`min(X)`	Return the minimum non-`NULL` value of all values in the group. The usual sort order is used to determine the minimum. `NULL` is only returned if all values in the group are `NULL`.
`sum(X)`, `total(X)`	Return the numeric sum of all non-`NULL` values in the group. If there are no non-`NULL` input rows, then `sum()` returns `NULL` but `total()` returns 0.0. `NULL` is not normally a helpful result for the sum of no rows but the SQL standard requires it and most other SQL database engines implement `sum()` that way so SQLite does it in the same way in order to be compatible. The nonstandard `total()` function is provided as a convenient way to work around this design problem in the SQL language.

The result of total() is always a floating-point value. The result of sum() is an integer value if all non-NULL inputs are integers and the sum is exact. If any input to sum() is neither an integer nor a NULL, or if an integer overflow occurs at any point during the computation, then sum() returns a floating-point value that might be an approximation to the true sum.

INSERT

```
sql-statement ::= INSERT [OR conflict-algorithm]
  INTO [database-name .] table-name [(column-list)] VALUES(value-list) |

  INSERT [OR conflict-algorithm] INTO [database-name .]
  table-name [(column-list)] select-statement
```

The INSERT statement comes in two basic forms. The first form (with the VALUES keyword) creates a single new row in an existing table. If no column-list is specified then the number of values must be the same as the number of columns in the table. If column-list is specified, then the number of values must match the number of specified columns. Columns of the table that do not appear in the column-list are filled with the default value, or with NULL if no default value is specified.

The second form of the INSERT statement takes its data from a SELECT statement. The number of columns in the result of the SELECT must exactly match the number of columns in the table if no column list is specified, or it must match the number of column named in the column list. A new entry is made in the table for every row of the SELECT result. The SELECT may be simple or compound. If the SELECT statement has an ORDER BY clause, the ORDER BY is ignored.

The optional conflict-clause allows the specification of an alternative constraint conflict resolution algorithm to use during this one command. See the section "ON CONFLICT" for additional information. For compatibility with MySQL, the parser allows the use of the single keyword REPLACE as an alias for INSERT OR REPLACE.

ON CONFLICT

```
conflict-clause ::= ON CONFLICT conflict-algorithm
conflict-algorithm ::= ROLLBACK | ABORT | FAIL | IGNORE | REPLACE
```

The ON CONFLICT clause is not a separate SQL command. It is a nonstandard clause that can appear in many other SQL commands. It is given its own section in the Appendix because it is not part of standard SQL and therefore might not be familiar.

The syntax for the ON CONFLICT clause is as shown for the CREATE TABLE command. For the INSERT and UPDATE commands, the keywords ON CONFLICT are replaced by OR, to make the syntax seem more natural. For example, instead of INSERT ON CONFLICT IGNORE we have INSERT OR IGNORE. The keywords change but the meaning of the clause is the same either way.

The ON CONFLICT clause specifies an algorithm used to resolve constraint conflicts. There are five choices: ROLLBACK, ABORT, FAIL, IGNORE, and REPLACE. The default algorithm is ABORT. Table A-4 shows what they mean.

When this conflict resolution strategy deletes rows in order to satisfy a constraint, it does not invoke delete triggers on those rows. But that may change in a future release.

Table A-4. *Conflict Resolution Algorithms*

Algorithm	Description
ROLLBACK	When a constraint violation occurs, an immediate ROLLBACK occurs, thus ending the current transaction, and the command aborts with a return code of SQLITE_CONSTRAINT. If no transaction is active (other than the implied transaction that is created on every command), then this algorithm works the same as ABORT.
ABORT	When a constraint violation occurs, the command backs out any prior changes it might have made and aborts with a return code of SQLITE_CONSTRAINT. But no ROLLBACK is executed so changes from prior commands within the same transaction are preserved. This is the default behavior.
FAIL	When a constraint violation occurs, the command aborts with a return code SQLITE_CONSTRAINT. But any changes to the database that the command made prior to encountering the constraint violation are preserved and are not backed out. For example, if an UPDATE statement encountered a constraint violation on the 100th row that it attempts to update, then the first 99 row changes are preserved but changes to rows 100 and beyond never occur.
IGNORE	When a constraint violation occurs, the one row that contains the constraint violation is not inserted or changed. But the command continues executing normally. Other rows before and after the row that contained the constraint violation continues to be inserted or updated normally. No error is returned.
REPLACE	When a UNIQUE constraint violation occurs, the preexisting rows that are causing the constraint violation are removed prior to inserting or updating the current row. Thus the insert or update always occurs. The command continues executing normally. No error is returned. If a NOT NULL constraint violation occurs, the NULL value is replaced by the default value for that column. If the column has no default value, then the ABORT algorithm is used.

The algorithm specified in the OR clause of an INSERT or UPDATE overrides any algorithm specified in a CREATE TABLE. If no algorithm is specified anywhere, the ABORT algorithm is used.

PRAGMA

sql-statement ::= **PRAGMA** *name* [= *value*] |
PRAGMA *function*(*arg*)

The PRAGMA command is a special command used to modify the operation of the SQLite library or to query the library for internal (non-table) data. The PRAGMA command is issued using the same interface as other SQLite commands (e.g., SELECT, INSERT) but is different in the following important respects:

- Specific pragma statements may be removed and others added in future releases of SQLite. Use with caution!
- No error messages are generated if an unknown pragma is issued. Unknown pragmas are simply ignored. This means that if there is a typo in a pragma statement the library does not inform the user of the fact.

- Some pragmas take effect during the SQL compilation stage, not the execution stage. This means if you are using the C-language `sqlite3_compile()`, `sqlite3_step()`, or `sqlite3_finalize()` API (or similar in a wrapper interface), the pragma may be applied to the library during the `sqlite3_compile()` call.
- The pragma command is not likely to be compatible with any other SQL engine.

The available pragmas fall into four basic categories:

- Pragmas used to query the schema of the current database
- Pragmas used to modify the library operation of the SQLite in some manner, or to query for the current mode of operation
- Pragmas used to query or modify the databases two version values, the `schema-version` and the `user-version`
- Pragmas used to debug the library and verify that database files are not corrupted

The pragmas that take an integer value also accept symbolic names. The strings "on", "true", and "yes" are equivalent to 1. The strings "off", "false", and "no" are equivalent to 0. These strings are case insensitive, and do not require quotes. An unrecognized string will be treated as 1, and will not generate an error. When the value is returned it is as an integer.

The following entries list all of the pragmas implemented in SQLite, arranged by category.

PRAGMA auto_vacuum (library operation)

```
PRAGMA auto_vacuum = 0 | 1;
```

Query or set the `auto_vacuum` flag in the database.

Normally, when a transaction that deletes data from a database is committed, the database file remains the same size. Unused database file pages are marked as such and reused later on, when data is inserted into the database. In this mode the `VACUUM` command is used to reclaim unused space.

When the `auto_vacuum` flag is set, the database file shrinks when a transaction that deletes data is committed. (The `VACUUM` command is not useful in a database with the `auto_vacuum` flag set.) To support this functionality the database stores extra information internally, resulting in slightly larger database files than would otherwise be possible.

It is only possible to modify the value of the `auto_vacuum` flag before any tables have been created in the database. No error message is returned if an attempt to modify the `auto_vacuum` flag is made after one or more tables have been created.

PRAGMA cache_size (library operation)

```
PRAGMA cache_size = Number-of-pages;
```

Query or change the maximum number of database disk pages that SQLite will hold in memory at once. Each page uses about 1.5K of memory. The default cache size is 2000. If you are doing `UPDATE`s or `DELETE`s that change many rows of a database and you do not mind if SQLite uses more memory, you can increase the cache size for a possible speed improvement.

When you change the cache size using the cache_size pragma, the change only endures for the current session. The cache size reverts to the default value when the database is closed and reopened. Use the default_cache_size pragma to check the cache size permanently.

PRAGMA case_sensitive_like (library operation)

```
PRAGMA case_sensitive_like = 0 | 1;
```

The default behavior of the LIKE operator is to ignore case for Latin1 characters. Hence, by default 'a' LIKE 'A' is true. The case_sensitive_like pragma can be turned on to change this behavior. When case_sensitive_like is enabled, 'a' LIKE 'A' is false but 'a' LIKE 'a' is still true.

PRAGMA count_changes (library operation)

```
PRAGMA count_changes = 0 | 1;
```

Query or change the count_changes flag. Normally, when the count_changes flag is not set, INSERT, UPDATE, and DELETE statements return no data. When count_changes is set, each of these commands returns a single row of data consisting of one integer value—the number of rows inserted, modified, or deleted by the command. The returned change count does not include any insertions, modifications, or deletions performed by triggers.

PRAGMA default_cache_size (library operation)

```
PRAGMA default_cache_size = Number-of-pages;
```

Query or change the maximum number of database disk pages that SQLite will hold in memory at once. Each page uses 1KB on disk and about 1.5KB in memory. This pragma works like the cache_size pragma with the additional feature that it changes the cache size persistently. With this pragma, you can set the cache size once and that setting is retained and reused every time you reopen the database.

PRAGMA default_synchronous (library operation)

This pragma was available in version 2.8 but was removed in version 3.0. It is a dangerous pragma whose use is discouraged. To help dissuade users of version 2.8 from employing this pragma, the documentation will not tell you what it does.

PRAGMA empty_result_callbacks (library operation)

```
PRAGMA empty_result_callbacks = 0 | 1;
```

Query or change the empty_result_callbacks flag.

The empty_result_callbacks flag affects the sqlite3_exec() API only. Normally, when the empty_result_callbacks flag is cleared, the callback function supplied to the sqlite3_exec() call is not invoked for commands that return zero rows of data. When empty_result_callbacks is set in this situation, the callback function is invoked exactly once, with the third parameter

set to 0 (NULL). This is to enable programs that use the sqlite3_exec() API to retrieve column names even when a query returns no data.

PRAGMA encoding (library operation)

```
PRAGMA encoding = "UTF-8";
PRAGMA encoding = "UTF-16";
PRAGMA encoding = "UTF-16le";
PRAGMA encoding = "UTF-16be";
```

If the main database has already been created, then this pragma returns the text encoding used by the main database, one of "UTF-8", "UTF-16le" (little-endian UTF-16 encoding), or "UTF-16be" (big-endian UTF-16 encoding). If the main database has not already been created, then the value returned is the text encoding that will be used to create the main database, if it is created by this session.

The second and subsequent forms of this pragma are only useful if the main database has not already been created. In this case the pragma sets the encoding that the main database will be created with if it is created by this session. The string "UTF-16" is interpreted as "UTF-16 encoding using native machine byte-ordering." If the second and subsequent forms are used after the database file has already been created, they have no effect and are silently ignored.

Once an encoding has been set for a database, it cannot be changed.

Databases created by the ATTACH command always use the same encoding as the main database.

PRAGMA full_column_names (library operation)

```
PRAGMA full_column_names = 0 | 1;
```

Query or change the full_column_names flag. This flag affects the way SQLite names columns of data returned by SELECT statements when the expression for the column is a table-column name or the wildcard (*). Normally, such result columns are named table-name/alias.column-name if the SELECT statement joins two or more tables together or simply column-name if the SELECT statement queries a single table. When the full_column_names flag is set, such columns are always named table-name/alias.column-name regardless of whether a join is performed.

If both the short_column_names and full_column_names are set, then the behavior associated with the full_column_names flag is exhibited.

PRAGMA fullfsync (library operation)

```
PRAGMA fullfsync = 0 | 1;
```

Query or change the fullfsync flag. This flag determines whether the F_FULLFSYNC syncing method is used on systems that support it. The default value is off. As of this writing only Mac OS X supports F_FULLFSYNC.

PRAGMA page_size (library operation)

```
PRAGMA page_size = bytes;
```

Query or set the page_size of the database. The page_size may only be set if the database has not yet been created. The page_size must be a power of 2 greater than or equal to 512 and less than or equal to 8192. The upper limit may be modified by setting the value of the macro SQLITE_MAX_PAGE_SIZE during compilation. The maximum upper bound is 32768.

PRAGMA read_uncommitted (library operation)

```
PRAGMA read_uncommitted = 0 | 1;
```

Query, set, or clear read uncommitted isolation. The default isolation level for SQLite is serializable. Any process or thread can select read uncommitted isolation, but serializable will still be used except between connections that share a common page and schema cache. Cache sharing is enabled using the sqlite3_enable_shared_cache() API and is only available between connections running the same thread. Cache sharing is off by default.

PRAGMA short_column_names (library operation)

```
PRAGMA short_column_names = 0 | 1;
```

Query or change the short_column_names flag. This flag affects the way SQLite names columns of data returned by SELECT statements when the expression for the column is a table or column name or the wildcard (*). Normally, such result columns are named table_name/alias.column_name if the SELECT statement joins two or more tables together or simply column_name if the SELECT statement queries a single table. When the short_column_names flag is set, such columns are always named column_name regardless of whether a join is performed.

If both the short_column_names and full_column_names are set, then the behavior associated with the full_column_names flag is exhibited.

PRAGMA synchronous (library operation)

```
PRAGMA synchronous = FULL; (2)
PRAGMA synchronous = NORMAL; (1)
PRAGMA synchronous = OFF; (0)
```

Query or change the setting of the synchronous flag. The first (query) form will return the setting as an integer. When synchronous is FULL (2), the SQLite database engine will pause at critical moments to make sure that data has actually been written to the disk surface before continuing. This ensures that if the operating system crashes or if there is a power failure, the database will be uncorrupted after rebooting. FULL synchronous is very safe, but it is also slow. When synchronous is NORMAL, the SQLite database engine will still pause at the most critical moments, but less often than in FULL mode. There is a very small (though non-zero) chance that a power failure at just the wrong time could corrupt the database in NORMAL mode. But in practice, you are more likely to suffer a catastrophic disk failure or some other unrecoverable hardware fault. With synchronous OFF (0), SQLite continues without pausing as soon as it has handed data off

to the operating system. If the application running SQLite crashes, the data will be safe, but the database might become corrupted if the operating system crashes or the computer loses power before that data has been written to the disk surface. On the other hand, some operations are as much as 50 or more times faster with synchronous OFF.

In SQLite version 2, the default value is NORMAL. For version 3, the default was changed to FULL.

PRAGMA temp_store (library operation)

```
PRAGMA temp_store = DEFAULT; (0)
PRAGMA temp_store = FILE; (1)
PRAGMA temp_store = MEMORY; (2)
```

Query or change the setting of the temp_store parameter. When temp_store is DEFAULT (0), the compile-time C preprocessor macro TEMP_STORE is used to determine where temporary tables and indexes are stored. When temp_store is MEMORY (2), temporary tables and indexes are kept in memory. When temp_store is FILE (1), temporary tables and indexes are stored in a file. The temp_store_directory pragma can be used to specify the directory containing this file. When the temp_store setting is changed, all existing temporary tables, indexes, triggers, and views are immediately deleted.

It is possible for the library compile-time C preprocessor symbol TEMP_STORE to override this pragma setting. Table A-5 summarizes the interaction of the TEMP_STORE preprocessor macro and the temp_store pragma.

Table A-5. *Temporary Storage Pragma Options*

TEMP_STORE	PRAGMA temp_store	Storage Used for TEMP Tables and Indexes
0	any	file
1	0	file
1	1	file
1	2	memory
2	0	memory
2	1	file
2	2	memory
3	any	memory

PRAGMA temp_store_directory (library operation)

```
PRAGMA temp_store_directory = 'directory-name';
```

Query or change the setting of the temp_store_directory—the directory where files used for storing temporary tables and indexes are kept. This setting lasts for the duration of the current connection only and resets to its default value for each new connection opened.

When the temp_store_directory setting is changed, all existing temporary tables, indexes, triggers, and viewers are immediately deleted. In practice, temp_store_directory should be set immediately after the database is opened.

The value directory-name should be enclosed in single quotes. To revert the directory to the default, set the directory-name to an empty string, for example, PRAGMA temp_store_directory =. An error is raised if directory-name is not found or is not writable.

The default directory for temporary files depends on the OS. For Unix/Linux/Mac OS X, the default is the first writable directory found in the list of /var/tmp, /usr/tmp, /tmp, and the current working directory. For Windows NT, the default directory is determined by Windows, generally C:\Documents and Settings\user-name\Local Settings\Temp\. Temporary files created by SQLite are unlinked immediately after opening, so that the operating system can automatically delete the files when the SQLite process exits. Thus, temporary files are not normally visible through ls or dir commands.

PRAGMA database_list (database schema)

For each open database, invoke the callback function once with information about that database. Arguments include the index and the name the database was attached with. The first row will be for the main database. The second row will be for the database used to store temporary tables.

PRAGMA foreign_key_list(table-name) (database schema)

For each foreign key that references a column in the argument table, invoke the callback function with information about that foreign key. The callback function will be invoked once for each column in each foreign key.

PRAGMA index_info(index-name) (database schema)

For each column that the named index references, invoke the callback function once with information about that column, including the column name, and the column number.

PRAGMA index_list(table-name) (database schema)

For each index on the named table, invoke the callback function once with information about that index. Arguments include the index name and a flag to indicate whether the index must be unique.

PRAGMA table_info(table-name) (database schema)

For each column in the named table, invoke the callback function once with information about that column, including the column name, data type, whether the column can be NULL, and the default value for the column.

PRAGMA [database.] (database version)

```
PRAGMA [database.]schema_version;
PRAGMA [database.]schema_version = integer ;
PRAGMA [database.]user_version;
PRAGMA [database.]user_version = integer ;
```

The pragmas schema_version and user_version are used to set or get the value of the schema_version and user_version, respectively. Both the schema_version and the user_version are 32-bit signed integers stored in the database header.

The schema_version is usually only manipulated internally by SQLite. It is incremented by SQLite whenever the database schema is modified (by creating or dropping a table or an index). The schema_version is used by SQLite each time a query is executed to ensure that the internal cache of the schema used when compiling the SQL query matches the schema of the database against which the compiled query is actually executed. Subverting this mechanism by using PRAGMA schema_version to modify the schema-version is potentially dangerous and may lead to program crashes or database corruption. Use with caution!

The user-version is not used internally by SQLite. It may be used by applications for any purpose.

PRAGMA integrity_check (debugging)

The command does an integrity check of the entire database. It looks for out-of-order records, missing pages, malformed records, and corrupt indexes. If any problems are found, then a single string is returned which is a description of all problems. If everything is in order, "ok" is returned.

PRAGMA parser_trace (debugging)

```
PRAGMA parser_trace = ON; (1)
PRAGMA parser_trace = OFF; (0)
```

Turn tracing of the SQL parser inside the SQLite library on and off. This is used for debugging. This only works if the library is compiled without the NDEBUG macro.

PRAGMA vdbe_trace (debugging)

```
PRAGMA vdbe_trace = ON; (1)
PRAGMA vdbe_trace = OFF; (0)
```

Turn tracing of the virtual database engine inside of the SQLite library on and off. This is used for debugging. See the VDBE documentation for more information.

PRAGMA vdbe_listing (debugging)

```
PRAGMA vdbe_listing = ON; (1)
PRAGMA vdbe_listing = OFF; (0)
```

Turn listings of virtual machine programs on and off. When listing is on, the entire contents of a program are printed just prior to beginning execution. This is like automatically executing an EXPLAIN prior to each statement. The statement executes normally after the listing is printed. This is used for debugging. See the VDBE documentation for more information.

REINDEX

```
sql-statement ::= REINDEX collation name
sql-statement ::= REINDEX [database-name .] table/index-name
```

The REINDEX command is used to delete and re-create indexes from scratch. This is useful when the definition of a collation sequence has changed.

In the first form, all indexes in all attached databases that use the named collation sequence are re-created. In the second form, if [database-name.]table/index-name identifies a table, then all indexes associated with the table are rebuilt. If an index is identified, then only this specific index is deleted and re-created.

If no database-name is specified and there exists both a table or an index and a collation sequence of the specified name, then indexes associated with the collation sequence only are reconstructed. This ambiguity may be dispelled by always specifying a database-name when reindexing a specific table or index.

REPLACE

```
sql-statement ::= REPLACE INTO [database-name .]
  table-name [( column-list )] VALUES ( value-list ) |

  REPLACE INTO [database-name .] table-name [( column-list )] select-statement
```

The REPLACE command is an alias for the INSERT OR REPLACE variant of the INSERT command. This alias is provided for compatibility with MySQL. See the INSERT command documentation for additional information.

ROLLBACK TRANSACTION

See “BEGIN TRANSACTION.”

SELECT

```
sql-statement ::= SELECT [ALL | DISTINCT] result [FROM table-list]
  [WHERE expr]
  [GROUP BY expr-list]
  [HAVING expr]
  [compound-op select]*
  [ORDER BY sort-expr-list]
  [LIMIT integer [( OFFSET | , ) integer]]

result ::= result-column [, result-column]*

result-column ::= * | table-name . * | expr [ [AS] string ]
```

table-list ::= *table* [*join-op table join-args*]*

table ::= *table-name* [**AS** *alias*] | (*select*) [**AS** *alias*]

join-op ::= , | [**NATURAL**] [**LEFT** | **RIGHT** | **FULL**] [**OUTER** | **INNER** | **CROSS**] **JOIN**

join-args ::= [**ON** *expr*] [**USING** (*id-list*)]

sort-expr-list ::= *expr* [*sort-order*] [, *expr* [*sort-order*]]*

sort-order ::= [**COLLATE** *collation-name*] [**ASC** | **DESC**]

compound_op ::= UNION | UNION ALL | INTERSECT | EXCEPT

The SELECT statement is used to query the database. The result of a SELECT is zero or more rows of data where each row has a fixed number of columns. The number of columns in the result is specified by the expression list in between the SELECT and FROM keywords. Any arbitrary expression can be used as a result. If a result expression is * then all columns of all tables are substituted for that one expression. If the expression is the name of a table followed by .* then the result is all columns in that one table.

The DISTINCT keyword causes a subset of result rows to be returned, in which each result row is different. NULL values are not treated as distinct from each other. The default behavior is that all result rows be returned, which can be made explicit with the keyword ALL.

The query is executed against one or more tables specified after the FROM keyword. If multiple table names are separated by commas, then the query is against the cross join of the various tables. The full SQL-92 join syntax can also be used to specify joins. A subquery in parentheses may be substituted for any table name in the FROM clause. The entire FROM clause may be omitted, in which case the result is a single row consisting of the values of the expression list.

The WHERE clause can be used to limit the number of rows over which the query operates.

The GROUP BY clause causes one or more rows of the result to be combined into a single row of output. This is especially useful when the result contains aggregate functions. The expressions in the GROUP BY clause do not have to be expressions that appear in the result. The HAVING clause is similar to WHERE except that HAVING applies after grouping has occurred. The HAVING expression may refer to values, even aggregate functions, that are not in the result.

The ORDER BY clause causes the output rows to be sorted. The argument to ORDER BY is a list of expressions that are used as the key for the sort. The expressions do not have to be part of the result for a simple SELECT, but in a compound SELECT each sort expression must exactly match one of the result columns. Each sort expression may be optionally followed by a COLLATE keyword and the name of a collating function used for ordering text and/or keywords ASC or DESC to specify the sort order.

The LIMIT clause places an upper bound on the number of rows returned in the result. A negative LIMIT indicates no upper bound. The optional OFFSET following LIMIT specifies how many rows to skip at the beginning of the result set. In a compound query, the LIMIT clause may only appear on the final SELECT statement. The limit is applied to the entire query not to the individual SELECT statement to which it is attached. Note that if the OFFSET keyword is used

in the LIMIT clause, then the limit is the first number and the offset is the second number. If a comma is used instead of the OFFSET keyword, then the offset is the first number and the limit is the second number. This seeming contradiction is intentional—it maximizes compatibility with legacy SQL database systems.

A compound SELECT is formed from two or more simple SELECTs connected by one of the operators UNION, UNION ALL, INTERSECT, or EXCEPT. In a compound SELECT, all the constituent SELECTs must specify the same number of result columns. There may be only a single ORDER BY clause at the end of the compound SELECT. The UNION and UNION ALL operators combine the results of the SELECTs to the right and left into a single big table. The difference is that in UNION all result rows are distinct where in UNION ALL there may be duplicates. The INTERSECT operator takes the intersection of the results of the left and right SELECTs. EXCEPT takes the result of left SELECT after removing the results of the right SELECT. When three or more SELECTs are connected into a compound, they group from left to right.

UPDATE

sql-statement ::= **UPDATE** [OR *conflict-algorithm*] [*database-name* .]
  table-name **SET** *assignment* [, *assignment*]* [**WHERE** *expr*]

assignment ::= *column-name* = *expr*

The UPDATE statement is used to change the value of columns in selected rows of a table. Each assignment in an UPDATE specifies a column name to the left of the equals sign and an arbitrary expression to the right. The expressions may use the values of other columns. All expressions are evaluated before any assignments are made. A WHERE clause can be used to restrict which rows are updated.

The optional conflict-clause allows the specification of an alternative conflict resolution algorithm to use during this one command. See the section "ON CONFLICT" for additional information.

VACUUM

sql-statement ::= **VACUUM** [*index-or-table-name*]

The VACUUM command is a SQLite extension modeled after a similar command found in PostgreSQL. If VACUUM is invoked with the name of a table or an index, then it is supposed to clean up the named table or index. In version 1.0 of SQLite, the VACUUM command would invoke gdbm_reorganize() to clean up the backend database file.

VACUUM became a no-op when the GDBM backend was removed from SQLite in version 2.0.0. VACUUM was reimplemented in version 2.8.1. The index or table name argument is now ignored.

When an object (table, index, or trigger) is dropped from the database, it leaves behind empty space. This makes the database file larger than it needs to be, but can speed up inserts. In time inserts and deletes can leave the database file structure fragmented, which slows down disk access to the database contents. The VACUUM command cleans the main database by copying its contents to a temporary database file and reloading the original database file from the copy.

This eliminates free pages, aligns table data to be contiguous, and otherwise cleans up the database file structure. It is not possible to perform the same process on an attached database file.

This command will fail if there is an active transaction. This command has no effect on an in-memory database.

As of SQLite version 3.1, an alternative to using the `VACUUM` command is autovacuum mode, enabled using the `auto_vacuum` pragma.

APPENDIX B

C API Reference

This appendix covers all functions in the SQLite version 3 API as covered in Chapter 6 and Chapter 7. Each function is indexed by its name, followed by its declaration, followed by the description of what it does.

Return Codes

Many of the API functions return integer result codes. There are 26 different return codes defined in the API, 23 of which correspond to errors. All of the SQLite return codes are listed in Table B-1. The API functions that return these codes are listed as follows:

`sqlite3_bind_`*`xxx`*`()`
`sqlite3_close()`
`sqlite3_create_collation()`
`sqlite3_collation_needed()`
`sqlite3_create_function()`
`sqlite3_prepare()`
`sqlite3_exec()`
`sqlite3_finalize()`
`sqlite3_get_table()`
`sqlite3_open()`
`sqlite3_reset()`
`sqlite3_step()`
`sqlite3_transfer_bindings()`

Table B-1. *SQLite Return Codes*

Code	Description
`SQLITE_OK`	The operation was successful.
`SQLITE_ERROR`	There is a general SQL error or missing database. It may be possible to obtain more error information depending on the error condition (`SQLITE_SCHEMA`, for example).
`SQLITE_PERM`	Access permission is denied. It is not possible to read or write to the database file.

Table B-1. *SQLite Return Codes (Continued)*

Code	Description
SQLITE_ABORT	A callback routine requested an abort.
SQLITE_BUSY	The database file is locked.
SQLITE_LOCKED	A table in the database is locked.
SQLITE_NOMEM	A call to malloc() has failed within a database operation.
SQLITE_READONLY	An attempt was made to write to a read-only database.
SQLITE_INTERRUPT	An operation was terminated by sqlite3_interrupt().
SQLITE_IOERR	Some kind of disk I/O error occurred.
SQLITE_CORRUPT	The database disk image is malformed. This will also occur if an attempt is made to open a non-SQLite database file as a SQLite database.
SQLITE_FULL	Insertion failed because the database is full. There is no more space on the file system, or the database file cannot be expanded.
SQLITE_CANTOPEN	SQLite is unable to open the database file.
SQLITE_PROTOCOL	There is a database lock protocol error.
SQLITE_EMPTY	(Internal only) The database table is empty.
SQLITE_SCHEMA	The database schema has changed.
SQLITE_CONSTRAINT	The operation is aborted due to constraint violation. This constant is returned if the SQL statement would have violated a database constraint (such as attempting to insert a value into a unique index that already exists in the index).
SQLITE_MISMATCH	There is a data type mismatch. An example of this is an attempt to insert noninteger 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	The library is used incorrectly. This error might occur if one or more of the SQLite API routines are used incorrectly. Examples of incorrect usage include calling sqlite3_exec() after the database has been closed using sqlite3_close() or calling sqlite3_exec() with the same database pointer simultaneously from two separate threads.
SQLITE_NOLFS	This value is returned if the SQLite library is compiled with large file support (LFS) enabled but LFS isn't supported on the host operating system.
SQLITE_AUTH	Authorization is denied. This occurs when a callback function installed using sqlite3_set_authorizer() returns SQLITE_DENY.
SQLITE_ROW	sqlite3_step() has another row ready.
SQLITE_DONE	sqlite3_step() has finished executing.

Glossary of Functions

sqlite3_aggregate_context

```
void *sqlite3_aggregate_context(sqlite3_context*, int nBytes);
```

Aggregate functions use this function to allocate a structure for storing state (for accumulating data). The first time this routine is called for a particular aggregate, a new structure of size `nBytes` is allocated, zeroed, and returned. On subsequent calls (for the same aggregate instance) the same buffer is returned. The buffer is automatically freed by SQLite when the aggregate is finalized.

sqlite3_bind

```
int sqlite3_bind_blob(sqlite3_stmt*, int, const void*, int n, void(*)(void*));
int sqlite3_bind_double(sqlite3_stmt*, int, double);
int sqlite3_bind_int(sqlite3_stmt*, int, int);
int sqlite3_bind_int64(sqlite3_stmt*, int, long long int);
int sqlite3_bind_null(sqlite3_stmt*, int);
int sqlite3_bind_text(sqlite3_stmt*, int, const char*, int n, void(*)(void*));
int sqlite3_bind_text16(sqlite3_stmt*, int, const void*, int n, void(*)(void*));

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

In a SQL statement, one or more literals can be replaced by a parameter (also called a *host parameter name*) of the forms `?`, `:name`, and `$var`. The parameter of the form `?` is called a *positional parameter*. The parameter of the form `:name` is called a *named parameter*, where `name` is an alphanumeric identifier. The parameter of the form `$var` is called a *TCL parameter*, where `var` is a variable name according to the syntax rules of the TCL programming language.

The `sqlite3_bind_`*xxx*`()` functions assign (or bind) values to parameters in a prepared SQL statement. (A prepared SQL statement is a string containing SQL commands passed to `sqlite3_prepare()` or `sqlite3_prepare16()`).

The first argument to `sqlite3_bind_`*xxx*`()` always is a pointer to the `sqlite3_stmt` structure returned from `sqlite3_prepare()`.

The second argument is the index of the parameter to be set. All parameters are identified by an index or a number. Positional parameters, for example, are numbered using sequential integer values, starting with 1 for the first parameter. Take for example the following statement that uses positional parameters:

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

The first positional parameter (represented by the first question mark) is assigned index 1; the second, index 2; and the third, index 3. When it comes time to bind values to these parameters, they will be identified in the `sqlite3_bind_`*xxx*`()` functions using their index values. The index values for named parameters can be looked up using the `sqlite3_bind_parameter_name()`. The third argument to `sqlite3_bind_`*xxx*`()` is the value to bind to the parameter.

In those sqlite3_bind_*xxx*() variants that have a fourth argument, its value is the number of bytes (or the size) of the value. This is the number of characters for UTF-8 strings and the number of bytes for UTF-16 strings and blobs. The number of bytes does not include the zero-terminator at the end of strings. If the fourth parameter is negative, the length of the string is computed using strlen().

The fifth argument to sqlite3_bind_blob(), sqlite3_bind_text(), and sqlite3_bind_text16() is a destructor used to dispose of the BLOB or TEXT after SQLite has finished with it. If the fifth argument is the special value SQLITE_STATIC, then the library assumes that the information is in static, unmanaged space and does not need to be freed. If the fifth argument has the value SQLITE_TRANSIENT, then SQLite makes its own private copy of the data before returning (and automatically cleans it up when the query is finalized).

The sqlite3_bind_*xxx*() routines must be called after sqlite3_prepare() or sqlite3_reset() and before sqlite3_step(). Bindings are not cleared by the sqlite3_reset() routine. Unbound parameters are interpreted as NULL.

sqlite3_bind_parameter_count

```
int sqlite3_bind_parameter_count(sqlite3_stmt*);
```

This function returns the number of parameters in the precompiled statement given as the argument.

sqlite3_bind_parameter_index

```
int sqlite3_bind_parameter_index(sqlite3_stmt*, const char *zName);
```

This function returns the index of the parameter with the given name (in zName). The name must match exactly. If there is no parameter with the given name, the function returns zero.

■ **Note** The string for the zName argument is always in UTF-8 encoding.

---

sqlite3_bind_parameter_name

```
const char *sqlite3_bind_parameter_name(sqlite3_stmt*, int n);
```

This function returns the name of the n^{th} parameter in the precompiled statement. Parameters of the form :name or $var have a name that is the string :name or $var. In other words, the initial : or $ is **included** as part of the name returned. Positional parameters (parameters of the form ?) have no name.

If the value for argument n is out of range or if the n^{th} parameter is nameless, then NULL is returned.

■ **Note** The returned string is always in UTF-8 encoding.

sqlite3_busy_handler

```
int sqlite3_busy_handler( sqlite3 *db,       /* db handle */
                          int(*)(void*,int), /* callback */
                          void*);            /* application data */
```

This function registers a callback function that can be invoked whenever an attempt is made by an API function to open a database table that another thread or process has locked. If a callback function is registered, SQLite may call it rather than having the API function return `SQLITE_BUSY`.

The callback function has two arguments. The first argument is a pointer to the application data passed in as the third argument to `sqlite3_busy_handler()`, and the second is the number of prior calls to the callback for the same lock. If the callback returns zero, then no additional attempts are made to access the database, and the blocked API call will return `SQLITE_BUSY`. If the callback returns non-zero, then another attempt is made by the API function to open the database for reading, and the cycle repeats itself.

If the callback argument in `sqlite3_busy_handler()` is set to `NULL`, then this will effectively remove any callback function (if one was previously registered). Then if an API function encounters a lock, it will immediately return `SQLITE_BUSY`.

The presence of a busy handler does not guarantee that it will be invoked whenever there is lock contention. If SQLite determines that invoking the busy handler could result in a deadlock, it will return `SQLITE_BUSY` instead. Consider a scenario where one process is holding a read lock that it is trying to promote to a reserved lock, and a second process is holding a reserved lock that it is trying to promote to an exclusive lock. The first process cannot proceed because it is blocked by the second and the second process cannot proceed because it is blocked by the first. If both processes invoke the busy handlers, neither will make any progress. Therefore, SQLite returns `SQLITE_BUSY` for the first process, hoping that this will induce the first process to release its read lock and allow the second process to proceed.

Since SQLite is re-entrant, the busy handler could in theory start a new query. However, the busy handler **may not close the database**. Closing the database from a busy handler will delete data structures out from under the executing query and will probably result in crashing the program.

sqlite3_busy_timeout

```
int sqlite3_busy_timeout( sqlite3 *db, /* db handle */
                          int ms);     /* time to sleep */
```

This function sets a busy handler that sleeps for a while when a table is locked. The handler will sleep multiple times until at least `ms` milliseconds have elapsed. After that time, the handler returns zero which causes the blocked API function to return `SQLITE_BUSY`. Calling this routine with an argument less than or equal to zero turns off all busy handlers.

sqlite3_changes

```
int sqlite3_changes(sqlite3*);
```

This function returns the number of database rows that were changed (or inserted or deleted) by the most recently completed INSERT, UPDATE, or DELETE statement. Only changes that are directly specified by the INSERT, UPDATE, or DELETE statement are counted. Auxiliary changes caused by triggers are not counted. To obtain the figure with these changes, use the sqlite3_total_changes() function. It provides the total number of changes **including** changes caused by triggers.

If called within a trigger, sqlite3_changes() reports the number of rows that were changed for the most recently completed INSERT, UPDATE, or DELETE statement within that trigger.

SQLite implements the command DELETE FROM table (without a WHERE clause) by dropping and recreating the table. This is an optimization that is much faster than going through and deleting individual records from the table. Because of this optimization, the change count for DELETE FROM table will be zero regardless of the number of records that were originally in the table. To obtain the actual number of deleted rows, use DELETE FROM table WHERE 1 instead, which disables the optimization.

sqlite3_clear_bindings

```
int sqlite3_clear_bindings(sqlite3_stmt*);
```

This function (re)sets all the parameters in the compiled SQL statement to NULL.

sqlite3_close

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

This function closes the connection given by db. If the operation is successful, the function returns SQLITE_OK. If the db argument is not a valid connection pointer returned by sqlite3_open(), or if the connection pointer has been closed previously, sqlite3_close() will return SQLITE_ERROR.

If there are prepared statements that have not been finalized, sqlite3_close() will return SQLITE_BUSY.

sqlite3_collation_needed

```
int sqlite3_collation_needed(
  sqlite3*,
  void*,
  void(*)(void*,sqlite3*,int eTextRep,const char*)
);

int sqlite3_collation_needed16(
  sqlite3 *db,  /* database handle  */
  void*,        /* application data */
  void(*crf)(void*,sqlite3*,int eTextRep,const void*)
);
```

This function allows the program to defer registration of custom collating sequences until they are actually needed at runtime.

It does this by registering a single callback function with the database handle whose job is to register undefined collation sequences (using `sqlite3_create_collation()`) when they are called upon at runtime.

From thereon out, when SQLite encounters an undefined collating sequence, it will call the callback function, pass it the name of the undefined collation sequence, and expect the callback to register a collating sequence by that name. Once complete, SQLite resumes the query requiring the custom collating sequence, using the function that the callback registered.

If `sqlite3_collation_needed16()` is used, the collating sequence name is passed as UTF-16 encoding in machine native byte order.

The callback function has the following declaration:

```
void crf( void*,        /* application data */
          sqlite3*,     /* db handle */
          int eTextRep, /* encoding */
          const void*)  /* collating sequence name */
```

The third argument is one of `SQLITE_UTF8`, `SQLITE_UTF16BE`, or `SQLITE_UTF16LE`, indicating the most desirable form of the collation sequence function required.

sqlite3_column

```
const void *sqlite3_column_blob(sqlite3_stmt*, int iCol);
int sqlite3_column_bytes(sqlite3_stmt*, int iCol);
int sqlite3_column_bytes16(sqlite3_stmt*, int iCol);
double sqlite3_column_double(sqlite3_stmt*, int iCol);
int sqlite3_column_int(sqlite3_stmt*, int iCol);
long long int sqlite3_column_int64(sqlite3_stmt*, int iCol);
const unsigned char *sqlite3_column_text(sqlite3_stmt*, int iCol);
const void *sqlite3_column_text16(sqlite3_stmt*, int iCol);
int sqlite3_column_type(sqlite3_stmt*, int iCol);
```

These functions return information about the value in a single column of the current row in a result set. In every case the first argument is a pointer to the SQL statement that is being executed (the statement handle returned from `sqlite3_prepare()`). The second argument (`iCol`) is the zero-based index of the column for which information should be returned. The left-most column has an index of zero.

If the statement handle is not currently pointing to a valid row, or if the column index is out of range, the result is undefined.

SQLite has five internal storage formats, also known as storage classes. These are defined in the API as follows:

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

The sqlite3_column_*xxx*() routines attempt to convert the internal format to the external *xxx* format requested in the respective function. For example, if the internal representation is FLOAT, and a TEXT result is requested (by sqlite3_column_text()), then sprintf() is used internally to do the conversion. SQLite's type conversion rules are listed in Table B-2.

Table B-2. *Type Conversion Rules*

Internal Type	Requested Type	Conversion
NULL	INTEGER	Result is 0
NULL	FLOAT	Result is 0.0
NULL	TEXT	Result is an empty string
NULL	BLOB	Result is a zero-length BLOB
INTEGER	FLOAT	Convert from integer to float
INTEGER	TEXT	ASCII rendering of the integer
INTEGER	BLOB	Same as for INTEGER to TEXT
FLOAT	INTEGER	Convert from float to integer
FLOAT	TEXT	ASCII rendering of the float
FLOAT	BLOB	Same as FLOAT to TEXT
TEXT	INTEGER	Use atoi()
TEXT	FLOAT	Use atof()
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

If the result is a BLOB then the sqlite3_column_bytes() routine returns the number of bytes in that BLOB. No type conversions occur. If the result is a string (or a number, since a number can be converted into a string) sqlite3_column_bytes() converts the value into a UTF-8 string and returns the number of bytes in the resulting string. The value returned does not include the NULL (\000) terminator at the end of the string. The sqlite3_column_bytes16() routine converts the value into UTF-16 encoding and returns the number of bytes (not characters) in the resulting string. The NULL (\u0000) terminator is not included in this count.

sqlite3_column_count

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

This function returns the number of columns in the result set returned by the prepared SQL statement. It returns zero if pStmt is an SQL statement that does not return data (an UPDATE statement for example).

sqlite3_column_database_name

```
const char *sqlite3_column_database_name(sqlite3_stmt *pStmt, int iCol);
const void *sqlite3_column_database_name16(sqlite3_stmt *pStmt, int iCol);
```

If the nth column returned by statement pStmt is a column reference, this function may be used to access the name of the database (either main, temp, or the name of an attached database) that contains the column. If the iColth column is not a column reference, NULL is returned.

See the description of function sqlite3_column_decltype() for exactly which expressions are considered column references.

Function sqlite3_column_database_name() returns a pointer to a UTF-8 encoded string. sqlite3_column_database_name16() returns a pointer to a UTF-16 encoded string.

sqlite3_column_decltype

```
const char *sqlite3_column_decltype(sqlite3_stmt *stmt, int iCol);
const void *sqlite3_column_decltype16(sqlite3_stmt*,int iCol);
```

This function returns the declared type of a column, as it is defined in the CREATE TABLE statement. The first argument is a statement handle (from a prepared SQL statement). The second argument is the column ordinal in the SQL statement. If the column does not correspond to an actual table column (but is, for example, a literal value, or the result of an expression) the function returns a NULL pointer.

The returned string is UTF-8 encoded for sqlite3_column_decltype() and UTF-16 encoded for sqlite3_column_decltype16(). For example, consider the following table:

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

If you compile the following statement

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

this routine would return the string INTEGER for the first column (iCol=0) and a NULL pointer for the second column (iCol=1).

sqlite3_column_name

```
const char *sqlite3_column_name(sqlite3_stmt*,int iCol);
const void *sqlite3_column_name16(sqlite3_stmt*,int iCol);
```

This function returns the column name in a prepared SQL statement. The first argument is the statement handle. The second argument is the column ordinal. The string returned is UTF-8 for sqlite3_column_name() and UTF-16 for sqlite3_column_name16().

sqlite3_column_origin_name

```
const char *sqlite3_column_origin_name(sqlite3_stmt *pStmt, int iCol);
const void *sqlite3_column_origin_name16(sqlite3_stmt *pStmt, int iCol);
```

If the nth column returned by statement pStmt is a column reference, these functions may be used to access the schema name of the referenced column in the database schema. If the iColth column is not a column reference, NULL is returned.

See the description of function sqlite3_column_decltype() for exactly which expressions are considered column references.

Function sqlite3_column_origin_name() returns a pointer to a UTF-8 encoded string. sqlite3_column_origin_name16() returns a pointer to a UTF-16 encoded string.

sqlite3_column_table_name

```
const char *sqlite3_column_table_name(sqlite3_stmt *pStmt, int iCol);
const void *sqlite3_column_table_name16(sqlite3_stmt *pStmt, int iCol);
```

If the iColth column returned by statement pStmt is a column reference. These functions may be used to access the name of the table that contains the column. If the nth column is not a column reference, NULL is returned.

See the description of function sqlite3_column_decltype() for exactly which expressions are considered column references.

Function sqlite3_column_table_name() returns a pointer to a UTF-8 encoded string. sqlite3_column_table_name16() returns a pointer to a UTF-16 encoded string.

sqlite3_column_type

```
int sqlite3_column_type(sqlite3_stmt*, int iCol);
```

Returns the storage class of a given column with ordinal iCol in the result set sqlite3_stmt. See also sqlite3_column.

sqlite3_commit_hook

```
void *sqlite3_commit_hook(
    sqlite3 *db,            /* db handle             */
    int(*xCallback)(void*), /* callback function ptr */
    void *pArg);            /* application data      */
```

This function registers a callback function (pointed to by xCallback) to be invoked whenever a new transaction is committed. The pArg argument is a pointer to application data which is passed back as the first and only argument of the callback function. If the callback function returns non-zero, then the commit is converted into a rollback.

If another function was previously registered, its pArg value is returned. Otherwise NULL is returned.

Registering a NULL function disables the callback. Only a single commit-hook callback can be registered at a time.

■ **Note** sqlite3_commit_hook() is currently marked as experimental. However, it is unlikely to change at this point since it has been in the API for so long.

---

sqlite3_complete

```
int sqlite3_complete(const char *sql);
int sqlite3_complete16(const void *sql);
```

These functions return true if the given input string sql comprises one or more complete SQL statements. The argument must be a NULL-terminated UTF-8 string for sqlite3_complete() and a NULL-terminated UTF-16 string for sqlite3_complete16().

sqlite3_create_collation

```
int sqlite3_create_collation(
  sqlite3*,
  const char *zName,
  int pref16,
  void*,
  int(*xCompare)(void*,int,const void*,int,const void*)
);

int sqlite3_create_collation16(
  sqlite3*,
  const char *zName,
  int pref16,
  void*,
  int(*xCompare)(void*,int,const void*,int,const void*)
);
```

These functions register new collation sequences.

The second argument (zName) is the name of the new collation sequence. It is provided as a UTF-8 string for sqlite3_create_collation() and as a UTF-16 string for sqlite3_create_collation16().

The third argument must be one of the constants defined as follows:

```
#define SQLITE_UTF8     1
#define SQLITE_UTF16BE  2
#define SQLITE_UTF16LE  3
#define SQLITE_UTF16    4
```

It specifies the encoding to use when passing strings to the xCompare functions. The SQLITE_UTF16 constant indicates that text strings are expected in UTF-16 in the native byte order of the host machine.

The fourth argument is a pointer to application-specific data that is passed back as the first argument in the xCompare function.

The fifth argument is the comparison routine (xCompare). If NULL provides for this argument, it deletes the collation sequence (so SQLite cannot call it anymore). The comparison function has the following form:

```
int xCompare( void*,            /* application data */
              int, const void*, /* string 1 info */
              int, const void*) /* string 2 info */
```

The remaining four arguments after the application data argument correspond to the two strings, each represented by a [length, data] pair and encoded in the encoding specified in the pref16 (third) argument of sqlite3_create_collation(). The comparison function should return negative, zero, or positive if the first string is less than, equal to, or greater than the second string, respectively (i.e., string1-string2).

sqlite3_create_function

```
int sqlite3_create_function(
  sqlite3 *,
  const char *zFunctionName,
  int nArg,
  int eTextRep,
  void *pUserData,
  void (*xFunc)(sqlite3_context*,int,sqlite3_value**),
  void (*xStep)(sqlite3_context*,int,sqlite3_value**),
  void (*xFinal)(sqlite3_context*)
);

int sqlite3_create_function16(
  sqlite3*,
  const void *zFunctionName,
  int nArg,
  int eTextRep,
  void *pUserData,
  void (*xFunc)(sqlite3_context*,int,sqlite3_value**),
  void (*xStep)(sqlite3_context*,int,sqlite3_value**),
  void (*xFinal)(sqlite3_context*)
);
#define SQLITE_UTF8     1
#define SQLITE_UTF16    2
#define SQLITE_UTF16BE  3
#define SQLITE_UTF16LE  4
#define SQLITE_ANY      5
```

These functions register custom SQL functions and aggregates implemented in C.

The first argument is the database handle to which the new function or aggregate is to be registered. If a single program uses more than one database handle internally, then user functions or aggregates must be added individually to each database handle with which they will be used.

The second argument (`zFunctionName`) is the function or aggregate name, as it will be addressed in SQL. It is encoded in UTF-8 for `sqlite3_create_function()` and UTF-16 for `sqlite3_create_function16()` (in fact this is the only way in which the two functions differ).

The third argument (`nArg`) is the number of arguments that the function or aggregate takes. If this `nArg` is -1, then the function or aggregate may take any number of arguments.

The fourth argument, `eTextRep`, specifies what type of text arguments this function prefers to receive. Any function should be able to work with UTF-8, UTF-16le, or UTF-16be. But some implementations may be more efficient with one representation than another. Users are allowed to specify separate implementations for the same function. A specific implementation is called depending on the text representation of the arguments. SQLite will select the implementation that provides the best match for the given situation. If there is only a single implementation that works for any text representation, the fourth argument should be set to `SQLITE_ANY`.

The fifth argument is a pointer to application-specific data. The function implementations can gain access to this pointer using the `sqlite3_user_data()` function.

The sixth, seventh, and eighth arguments (`xFunc`, `xStep` and `xFinal`) are pointers to user-implemented C functions that implement the user function or aggregate. User-defined functions must provide an implementation for the `xFunc` callback only, and pass `NULL` pointers to `xStep` and `xFinal` arguments. Likewise, user-defined aggregates must provide implementations for `xStep` and `xFinal`, and pass `NULL` for `xFunc`.

If `NULL` is passed in all three function callbacks, it will delete (or unregister) the existing user-defined function or aggregate by that name.

`sqlite3_create_function()` will return `SQLITE_ERROR` if there is an inconsistent set of callback values specified, such as an `xFunc` and an `xFinal`, or an `xStep` but no `xFinal`.

sqlite3_data_count

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

This function returns the number of values in the current row of the result set.

After a call to `sqlite3_step()` which returns `SQLITE_ROW`, `sqlite3_data_count()` will return the same value as the `sqlite3_column_count()` function.

If called before `sqlite3_step()` has been called on a prepared SQL statement, `sqlite3_data_count()` will return zero. Likewise, if called after `sqlite3_step()` has returned an `SQLITE_DONE`, `SQLITE_BUSY`, or an error code, `sqlite3_data_count()` will also return zero.

sqlite3_db_handle

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

This function returns the database handle corresponding to the prepared statement handle provided as the first (and only) argument. That is, this is the database handle used by `sqlite3_prepare()` to create the statement handle provided as the argument.

This function comes in handy when implementing query processing functions that have access to the statement handle, but not the database handle. If errors arise in the query processing, the query processing function will need the database handle in order to call `sqlite3_errmsg()` or `sqlite3_errcode()` for additional error information. This is where `sqlite3_db_handle()` helps out.

sqlite3_enable_shared_cache

```
int sqlite3_enable_shared_cache(int val);
```

This routine enables or disables the sharing of the database cache and schema data structures between connections to the same database. Sharing is enabled if the argument is true (`val!=0`) and disabled if the argument is false (`val=0`).

Cache sharing is enabled and disabled on a thread-by-thread basis. Each call to this routine enables or disables cache sharing only for connections created in the same thread in which this routine is called. There is no mechanism for sharing cache between database connections running in different threads.

Sharing must be disabled prior to shutting down a thread, or else the thread will leak memory. Call this routine with an argument of zero to turn off sharing. Or use the `sqlite3_thread_cleanup()` API.

This routine must not be called when any database connections are active in the current thread. Enabling or disabling shared cache while there are active database connections will result in memory corruption.

When the shared cache is enabled, the following routines must always be called from the same thread: `sqlite3_open()`, `sqlite3_prepare()`, `sqlite3_step()`, `sqlite3_reset()`, `sqlite3_finalize()`, and `sqlite3_close()`. This is due to the fact that the shared cache makes use of thread-specific storage so it will be available for sharing with other connections.

This routine returns `SQLITE_OK` if shared cache is enabled or disabled successfully. An error code is returned otherwise.

Shared cache is disabled by default for backward compatibility.

sqlite3_errcode

```
int sqlite3_errcode(sqlite3 *db);
```

This function returns the error code for the most recently failed API call associated with the database handle provided in the argument. If a prior API call fails but the most recent API call succeeds, the return value from this function is undefined.

Calls to many API functions set the error code and string returned by `sqlite3_errcode()`, `sqlite3_errmsg()`, and `sqlite3_errmsg16()` (overwriting the previous values). Note that calls to `sqlite3_errcode()`, `sqlite3_errmsg()`, and `sqlite3_errmsg16()` themselves do not affect the results of future invocations. Calls to API routines that do not return an error code (for example, `sqlite3_data_count()` or `sqlite3_mprintf()`) do not change the error code returned by this routine.

Assuming no other intervening API calls are made, the error code returned by this function is associated with the same error as the strings returned by `sqlite3_errmsg()` and `sqlite3_errmsg16()`.

sqlite3_errmsg

```
const char *sqlite3_errmsg(sqlite3*);
const void *sqlite3_errmsg16(sqlite3*);
```

This function returns a pointer to a UTF-8 encoded string (`sqlite3_errmsg`) or a UTF-16 encoded string (`sqlite3_errmsg16`) describing in English the error condition for the most recent API call. The returned string is always terminated by a `0x00` byte.

The string `not an error` is returned when the most recent API call is successful.

sqlite3_exec

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

This function executes one or more statements of SQL contained in the second (`sql`) argument.

If one or more of the SQL statements are queries, then the callback function specified by the third argument is invoked once for each row of the query result. This callback should normally return zero. If the callback returns a non-zero value then the query is aborted, all subsequent SQL statements are skipped, and the `sqlite3_exec()` function returns the `SQLITE_ABORT`.

The fourth argument is an arbitrary pointer to application-specific data that is passed back to the callback function as its first argument.

The second argument to the callback function is the number of columns in the query result. The third argument to the callback is an array of strings holding the values for each column. The fourth argument to the callback is an array of strings holding the names of each column.

The callback function may be `NULL`, even for queries. A `NULL` callback is not an error. It just means that no callback will be invoked.

If an error occurs while parsing or evaluating the SQL (but not while executing the callback) then an appropriate error message is written into memory obtained from `malloc()`, and `errmsg` is made to point to that message. The calling function is responsible for freeing the memory that holds the error message, using `sqlite3_free()`. If `errmsg` is set to `NULL`, then no error message will be provided.

If all SQL commands succeed, the function returns `SQLITE_OK`, otherwise it will return the appropriate error code. The particular return value depends on the type of error. If the query could not be executed because a database file is locked or busy, then this function returns `SQLITE_BUSY`. This behavior can be modified somewhat using the `sqlite3_busy_handler()` and `sqlite3_busy_timeout()` functions.

sqlite3_expired

```
int sqlite3_expired(sqlite3_stmt*);
```

This function returns true (non-zero) if the statement supplied as an argument needs to be recompiled. A statement needs to be recompiled whenever the execution environment changes in a way that would alter the program that sqlite3_prepare() generates—for example, if new functions or collating sequences are registered, or if an authorizer function is added or changed.

sqlite3_finalize

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

This function deletes a prepared SQL statement obtained by a previous call to sqlite3_prepare() or sqlite3_prepare16(). If the statement is executed successfully, or not executed at all, then SQLITE_OK is returned. If execution of the statement fails then the corresponding error code is returned.

All prepared statements must be finalized before sqlite3_close() is called, otherwise sqlite3_close() will fail and return SQLITE_BUSY.

This function can be called at any point during the execution of the virtual machine. If the virtual machine has not completed execution when this routine is called, it is like encountering an error or an interrupt (see sqlite3_interrupt()). In this case, incomplete updates may be rolled back and transactions canceled, depending on the circumstances, and sqlite3_finalize() will return SQLITE_ABORT.

sqlite3_free

```
void sqlite3_free(char *z);
```

This function frees memory obtained from sqlite3_mprintf(), sqlite3_vmprintf(), or error messages generated from sqlite3_exec().

sqlite3_get_table, sqlite3_free_table

```
int sqlite3_get_table(
  sqlite3*,              /* An open database */
  const char *sql,       /* SQL to be executed */
  char ***resultp,       /* Result written to a char *[]  that this points to */
  int *nrow,             /* Number of result rows written here */
  int *ncolumn,          /* Number of result columns written here */
  char **errmsg          /* Error msg written here */
);
void sqlite3_free_table(char **result);
```

This function is just a wrapper around sqlite3_exec(). Instead of invoking a user-supplied callback function for each row of the result, this function stores each row of the result in memory obtained from malloc(), and returns the entire result of the query.

For example, suppose the query result were the following:

```
Name        | Age
------------+----------
Alice       | 43
Bob         | 28
Cindy       | 21
```

If the third argument is `&azResult`, for example, after `sqlite3_get_table()` returns, `azResult` will contain the following data:

```
azResult[0] = "Name";
azResult[1] = "Age";
azResult[2] = "Alice";
azResult[3] = "43";
azResult[4] = "Bob";
azResult[5] = "28";
azResult[6] = "Cindy";
azResult[7] = "21";
```

Notice that there is an extra row of data containing the column headers. But the `*nrow` return value is still 3. `*ncolumn` is set to 2. In general, the number of values inserted into `azResult` will be `((*nrow) + 1)*(*ncolumn)`.

After the calling function has finished using the result, it should pass the result data pointer to `sqlite3_free_table()` in order to release the memory that was allocated. Because of the way the `malloc()` happens, the calling function must not try to free the memory itself, but rather call `sqlite3_free_table()` to release the memory properly and safely.

The return value of this routine is the same as from `sqlite3_exec()`.

sqlite3_get_autocommit

```
int sqlite3_get_autocommit(sqlite3*);
```

This function returns whether or not the database handle provided as the argument is in auto-commit mode. It returns true (non-zero) if it is and false (zero) if not.

By default, auto-commit mode is on. Auto-commit is disabled by a `BEGIN` statement and re-enabled by the next `COMMIT` or `ROLLBACK`.

sqlite3_global_recover

```
int sqlite3_global_recover();
```

This function is called to recover from a `malloc()` failure within the SQLite library. Normally, after a single `malloc()` fails the library refuses to function (all major calls return `SQLITE_NOMEM`). This function restores the library state so it can be used again.

All existing statement handles must be finalized or reset before this call is made. Otherwise, SQLITE_BUSY is returned. If any in-memory databases are in use, either as a main or a temp database, SQLITE_ERROR is returned. In either of these cases, the library is not reset and remains unusable.

This function is **not** thread safe. Calling this from within a threaded application when threads other than the caller have used SQLite is dangerous and will almost certainly result in malfunctions.

This functionality can be omitted from a build by defining the SQLITE_OMIT_GLOBALRECOVER at compile time.

sqlite3_interrupt

```
void sqlite3_interrupt(sqlite3*);
```

This function causes any pending database operation on a given database handle to abort and return at its earliest opportunity. This routine is typically called in response to a user action such as pressing Cancel or CTRL-C where the user wants a long query operation to halt immediately.

sqlite3_last_insert_rowid

```
long long int sqlite3_last_insert_rowid(sqlite3*);
```

This function returns the autoincrement primary key value generated from the last successful INSERT statement.

Each entry in a SQLite table has a unique integer key. (The key is the value of the INTEGER PRIMARY KEY column if there is such a column, otherwise the key is generated at random. The unique key is always available as the ROWID, OID, or _ROWID_ column.)

This function is similar to the mysql_insert_id() function from MySQL.

sqlite3_libversion

```
const char *sqlite3_libversion(void);
```

This function returns a pointer to a string, which contains the version number of the library. The same string is available in the global variable named sqlite3_version within SQL. This interface is provided since Microsoft Windows is unable to access global variables in DLLs.

sqlite3_mprintf

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

These functions are variants of the sprintf() from the standard C library. The resulting string is written into memory obtained from malloc() so that there is never a possibility of buffer

overflow. These routines also implement some additional formatting options that are useful for constructing SQL statements.

The strings returned by these routines should be freed by calling `sqlite3_free()`.

All of the usual `printf()` formatting options apply. In addition, there is a `%q` option. This option works like `%s` in that it substitutes a `NULL`-terminated string from the argument list. But `%q` also doubles every single quote character (`'`). The `%q` option is designed for use inside a string literal. By doubling each single quote character, it escapes that character and allows it to be inserted into the string.

For example, say some string variable contains the following text:

```
char *zText = "It's a happy day!";
```

One can use this text in a SQL statement as follows:

```
sqlite3_exec_printf( db, "INSERT INTO table VALUES('%q')",
                     callback1, 0, 0, zText);
```

Because the `%q` format string is used, the single quote character in `zText` is escaped and the SQL generated is as follows:

```
INSERT INTO table1 VALUES('It''s a happy day!')
```

This is correctly formatted SQL. Had we used `%s` instead of `%q`, the generated SQL would have looked like this:

```
INSERT INTO table1 VALUES('It's a happy day!');
```

This would result in a SQL syntax error. As a general rule, you should always use `%q` instead of `%s` when inserting text into a string literal.

sqlite3_open

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

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

This function opens the SQLite database file specified in `filename`. The `filename` argument is UTF-8 encoded for `sqlite3_open()` and UTF-16 encoded in the native byte order for `sqlite3_open16()`. A database handle is returned in the `ppDb` argument even if an error occurs. If the database is opened (or created) successfully, then `SQLITE_OK` is returned. Otherwise an error code is returned. The `sqlite3_errmsg()` or `sqlite3_errmsg16()` routines can be used to obtain an English language description of the error.

If the database file does not exist, then a new database will be created as needed. The encoding for the database will be UTF-8 if `sqlite3_open()` is called and UTF-16 if `sqlite3_open16` is used.

Resources associated with the database handle should be released by passing it to `sqlite3_close()` when it is no longer required whether or not an error occurs when it is opened.

sqlite3_prepare

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

int sqlite3_prepare16(
  sqlite3 *db,            /* Database handle */
  const void *zSql,       /* SQL statement, UTF-16 encoded */
  int nBytes,             /* Length of zSql in bytes. */
  sqlite3_stmt **ppStmt,  /* OUT: Statement handle */
  const void **pzTail     /* OUT: Pointer to unused portion of zSql */
);
```

This function prepares a SQL statement for execution by compiling it into a byte-code program readable by the SQLite virtual machine.

The first argument is an open SQLite database handle.

The second argument (`zSql`) is the statement to be compiled. The only difference between the two functions is that this argument is assumed to be encoded in UTF-8 for the `sqlite3_prepare()` and UTF-16 for `sqlite3_prepare16()`.

If the third argument (`nBytes`) is less than zero, then `zSql` is read up to the first `NULL` terminator. If `nBytes` is not less than zero, then it should be the length of the `zSql` string in bytes (not characters).

The `pzTail` argument is made to point to the first byte past the end of the first SQL statement in `zSql`. This routine only compiles the first statement in `zSql`, so `*pzTail` is left pointing to what remains uncompiled.

The fourth argument (`*ppStmt`) is left pointing to a compiled SQL statement that can be executed using `sqlite3_step()`. If there is an error, `*ppStmt` may be set to `NULL`. If the input text contained no SQL (if the input is an empty string or a comment) then `*ppStmt` is set to `NULL`. The calling procedure is responsible for deleting this compiled SQL statement using `sqlite3_finalize()` after it has finished with it.

On success, `sqlite3_prepare()` returns `SQLITE_OK`. Otherwise the appropriate error code is returned.

sqlite3_progress_handler

```
void sqlite3_progress_handler(sqlite3*, int n, int(*)(void*), void*);
```

This function configures a callback function (the progress callback) that is invoked periodically during long running calls to `sqlite3_exec()`, `sqlite3_step()`, and `sqlite3_get_table()`. An example usage of this API would be to keep a GUI updated during a large query.

The progress callback is invoked once for every `n` virtual machine opcode, where `n` is the second argument to this function. The progress callback itself is identified by the third argument to this function. The fourth argument to this function is a void pointer passed to the progress callback function each time it is invoked.

If a call to `sqlite3_exec()`, `sqlite3_step()`, or `sqlite3_get_table()` results in less than `n` opcodes being executed, then the progress callback is not invoked.

To remove the progress callback altogether, pass `NULL` as the third argument to this function.

If the progress callback returns a result other than zero, the current query is immediately terminated and any database changes are rolled back. If the query is part of a larger transaction, the transaction is not rolled back and remains active. The `sqlite3_exec()` call returns `SQLITE_ABORT`.

■ **Note** `sqlite3_progress_handler()` is currently marked as experimental. However, it is unlikely to change at this point since it has been in the API for so long.

---

sqlite3_reset

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

This function resets a prepared SQL statement obtained by a previous call to `sqlite3_prepare()` or `sqlite3_prepare16()` back to its initial state. The statement handle is then ready to be re-executed. Any SQL parameters that have values bound to them will retain their values on the subsequent execution.

sqlite3_result_*xxx*

```
void sqlite3_result_blob(sqlite3_context*, const void*, int n, void(*)(void*));
void sqlite3_result_double(sqlite3_context*, double);
void sqlite3_result_error(sqlite3_context*, const char*, int);
void sqlite3_result_error16(sqlite3_context*, const void*, int);
void sqlite3_result_int(sqlite3_context*, int);
void sqlite3_result_int64(sqlite3_context*, long long int);
void sqlite3_result_null(sqlite3_context*);
void sqlite3_result_text(sqlite3_context*, const char*, int n, void(*)(void*));
void sqlite3_result_text16(sqlite3_context*, const void*, int n, void(*)(void*));
void sqlite3_result_text16be(sqlite3_context*, const void*, int n, void(*)(void*));
void sqlite3_result_text16le(sqlite3_context*, const void*, int n, void(*)(void*));
void sqlite3_result_value(sqlite3_context*, sqlite3_value*);
```

These functions are used to set the return value for user-defined functions. The sqlite3_result_value() routine is used to return an exact copy of one of the arguments as the function return value.

The operation of these routines is very similar to the operation of sqlite3_bind_blob() and its cousins. Refer to the documentation of those functions for additional information.

sqlite3_rollback_hook

```
void *sqlite3_rollback_hook(sqlite3*, void(*)(void *), void*);
```

Register a callback to be invoked whenever a transaction is rolled back.

The new callback function overrides any existing rollback-hook callback. If there was an existing callback, then its pArg value (the third argument to sqlite3_rollback_hook() when it was registered) is returned. Otherwise, NULL is returned.

For the purposes of this API, a transaction is said to have been rolled back if an explicit ROLLBACK statement is executed, or an error or constraint causes an implicit rollback to occur. The callback is not invoked if a transaction is automatically rolled back because the database connection is closed.

sqlite3_set_authorizer

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

This function registers a callback with the SQLite library, which can be used to monitor and control database events. The callback is invoked (at query compile-time, not at run-time) for each attempt to access a column of a table in the database. The callback should return SQLITE_OK if access is allowed, SQLITE_DENY if the entire SQL statement should be aborted with an error, and SQLITE_IGNORE if the column should be treated as a NULL value.

The second argument to the access authorization function will be one of the defined constants shown. These values signify the kind of operation to be authorized. The third and fourth arguments to the authorization function will be arguments or NULL, depending on which of the following codes is used as the second argument. These are listed in Table B-3.

Table B-3. *SQLite Authorization Events*

Event Code	Argument 3	Argument 4
SQLITE_CREATE_INDEX	Index name	Table name
SQLITE_CREATE_TABLE	Table name	NULL
SQLITE_CREATE_TEMP_INDEX	Index name	Table name
SQLITE_CREATE_TEMP_TABLE	Table name	NULL

Table B-3. *SQLite Authorization Events*

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

The fifth argument is the name of the database (main, temp, etc.) if applicable. The sixth argument is the name of the inner-most trigger or view that is responsible for the access attempt, or NULL if this access attempt is directly from input SQL code.

The return value of the authorization function should be one of the constants SQLITE_OK, SQLITE_DENY, or SQLITE_IGNORE.

The intent of this routine is to allow applications to safely execute user-entered SQL. An appropriate callback can deny the user-entered SQL access to certain operations (e.g., anything that changes the database), or to deny access to certain tables or columns within the database.

sqlite3_sleep

```
int sqlite3_sleep(int ms);
```

This function sleeps for a little while. The argument is the number of milliseconds to sleep.

If the operating system does not support sleep requests with millisecond time resolution, the time will be rounded up to the nearest second. The number of milliseconds of sleep actually requested from the operating system is returned.

sqlite3_soft_heap_limit

```
void sqlite3_soft_heap_limit(int n);
```

This routine sets the soft heap limit for the current thread to `n`. If the total heap usage by SQLite in the current thread exceeds `n`, then `sqlite3_release_memory()` is called to try to reduce the memory usage below the soft limit.

Prior to shutting down a thread, `sqlite3_soft_heap_limit()` must be set to zero (the default) or else the thread will leak memory. Alternatively, use the `sqlite3_thread_cleanup()` API.

A negative or zero value for `n` means that there is no soft heap limit and `sqlite3_release_memory()` will only be called when memory is exhausted. The default value for the soft heap limit is zero.

SQLite makes a best effort to honor the soft heap limit. But if it is unable to reduce memory usage below the soft limit, execution will continue without error or notification. This is why the limit is called a "soft" limit. It is advisory only.

This routine is only available if memory management has been enabled by compiling with the `SQLITE_ENABLE_MEMORY_MANAGMENT` macro.

sqlite3_step

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

This function executes a prepared query.

After a SQL query has been prepared with a call to either `sqlite3_prepare()` or `sqlite3_prepare16()`, this function must be called one or more times to execute the statement. The return value will be either `SQLITE_BUSY`, `SQLITE_DONE`, `SQLITE_ROW`, `SQLITE_ERROR`, or `SQLITE_MISUSE`.

`SQLITE_BUSY` means that the database engine attempted to open a locked database and there is no busy callback registered. Call `sqlite3_step()` again to retry the open.

`SQLITE_DONE` means that the statement has finished executing successfully. `sqlite3_step()` should not be called again on this virtual machine without first calling `sqlite3_reset()` to reset the virtual machine back to its initial state.

If the SQL statement being executed returns any data, then `SQLITE_ROW` is returned each time a new row of data is ready for processing by the caller. The values may be accessed using the `sqlite3_column()` functions. Subsequent rows are retrieved by calling `sqlite3_step()`.

SQLITE_ERROR means that a run-time error (such as a constraint violation) has occurred. sqlite3_step() should not be called again in this situation. More information on the error may be found by calling sqlite3_errmsg().

SQLITE_MISUSE means that the function was called inappropriately. This could happen if it was called with a statement handle that had already been finalized or with one that had previously returned SQLITE_ERROR or SQLITE_DONE. Or it could be the case that the same database connection is being used simultaneously by two or more threads.

sqlite3_column_meta_data

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

This routine is used to obtain meta information about a specific column of a specific database table accessible using the connection handle passed as the first function argument.

The column is identified by the second, third, and fourth parameters to this function. The second parameter is either the name of the database (i.e., main, temp, or an attached database) containing the specified table, or NULL. If it is NULL, all attached databases are searched for the table, using the same algorithm the database engine uses to resolve unqualified table references.

The third and fourth parameters to this function are the table and column name of the desired column, respectively. Neither of these parameters may be NULL.

Meta information is returned by writing to the memory locations passed as the fifth and subsequent parameters to this function, which are defined in Table B-4. Any of these arguments may be NULL, in which case the corresponding element of meta information is omitted.

Table B-4. *Out Parameters for sqlite3_column_meta_data*

Parameter	Output Type	Description
5	const char*	Declared data type
6	const char*	Name of the column's default collation sequence
7	int	True if the column has a NOT NULL constraint
8	int	True if the column is part of the PRIMARY KEY
9	int	True if the column is AUTOINCREMENT

The memory pointed to by the character pointers returned for the declaration type and collation sequence is valid only until the next call to any SQLite API function.

This function may load one or more schemas from database files. If an error occurs during this process, or if the requested table or column cannot be found, an error code is returned and an error message is left in the database handle (to be retrieved using `sqlite3_errmsg()`). Specifying an SQL view instead of a table as the third argument is also considered an error.

If the specified column is `ROWID`, `OID`, or `_ROWID_` and an `INTEGER PRIMARY KEY` column has been explicitly declared, the output parameters are set for the explicitly declared column. If there is no explicitly declared `INTEGER PRIMARY KEY` column, then the data-type is `INTEGER`, the collation sequence is `BINARY`, and the primary-key flag is set. Both the not-`NULL` and the autoincrement flags are clear.

This API is only available if the library is compiled with the `SQLITE_ENABLE_COLUMN_METADATA` preprocessor directive defined.

sqlite3_thread_cleanup

```
void sqlite3_thread_cleanup(void);
```

This routine ensures that a thread that has used SQLite in the past has released any thread-local storage it might have allocated. When the rest of the API is used properly, the cleanup of thread-local storage should be completely automatic. You should never really need to invoke this API. It is provided to you as a precaution and as a potential workaround for future thread-related memory leaks.

sqlite3_total_changes

```
int sqlite3_total_changes(sqlite3*);
```

This function returns the total number of database rows that have to be modified, inserted, or deleted, since the database connection was created using `sqlite3_open()`. All changes are counted, including changes by triggers, and changes to temp and auxiliary databases. Changes to the `sqlite_master` table (caused by statements such as `CREATE TABLE`) are not counted. Changes counted when an entire table is deleted using `DROP TABLE` are not counted either.

SQLite implements the command `DELETE FROM table` (without a `WHERE` clause) by dropping and recreating the table. This is much faster than going through and deleting individual elements from the table. Because of this optimization, the change count for `DELETE FROM table` will be zero regardless of the number of elements that were originally in the table. To get an accurate count of the number of rows deleted, use `DELETE FROM table WHERE 1` instead.

See also `sqlite3_changes()`.

sqlite3_trace

```
void *sqlite3_trace(
    sqlite3*,                         /* database handle   */
    void(*xTrace)(void*,const char*), /* callback function */
    void*);                           /* application data  */
```

This function registers a callback function that will be called each time an SQL statement is evaluated on a given connection. The callback function is invoked on the first call to `sqlite3_step()`, after calls to `sqlite3_prepare()` or `sqlite3_reset()`. This function can be used (for example) to generate a log file of all SQL executed against a database. This can be useful when debugging an application that uses SQLite.

sqlite3_transfer_bindings

```
int sqlite3_transfer_bindings(sqlite3_stmt*, sqlite3_stmt*);
```

This function moves all bindings from the first prepared statement over to the second. This function is useful, for example, if the first prepared statement fails with an `SQLITE_SCHEMA` error. In this case, the same SQL can be prepared in the second statement. Then all of the bindings can be transferred to that statement before the first statement is finalized.

sqlite3_update_hook

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

Register a callback function with the database connection identified by the first argument to be invoked whenever a row is updated, inserted, or deleted. Any callback set by a previous call to this function for the same database connection is overridden.

The second argument is a pointer to the function to invoke when a row is updated, inserted, or deleted. The first argument to the callback is a copy of the third argument to `sqlite3_update_hook`. The second callback argument is one of `SQLITE_INSERT`, `SQLITE_DELETE`, or `SQLITE_UPDATE`, depending on the operation that caused the callback to be invoked. The third and fourth arguments to the callback contain pointers to the database and table name containing the affected row. The final callback parameter is the `ROWID` of the row. In the case of an update, this is the `ROWID` after the update takes place.

The update hook is not invoked when internal system tables are modified (i.e., `sqlite_master` and `sqlite_sequence`).

If another function was previously registered, its `pArg` value is returned. Otherwise `NULL` is returned.

See also `sqlite3_commit_hook()` and `sqlite3_rollback_hook()`.

sqlite3_user_data

```
void *sqlite3_user_data(sqlite3_context*);
```

This function returns the application data specific to a user-defined function or aggregate. The data returned corresponds to the `pUserData` argument provided to `sqlite3_create_function()` or `sqlite3_create_function16()` when the function or aggregate was registered.

sqlite3_value_*xxx*

```
const void *sqlite3_value_blob(sqlite3_value*);
int sqlite3_value_bytes(sqlite3_value*);
int sqlite3_value_bytes16(sqlite3_value*);
double sqlite3_value_double(sqlite3_value*);
int sqlite3_value_int(sqlite3_value*);
long long int sqlite3_value_int64(sqlite3_value*);
const unsigned char *sqlite3_value_text(sqlite3_value*);
const void *sqlite3_value_text16(sqlite3_value*);
const void *sqlite3_value_text16be(sqlite3_value*);
const void *sqlite3_value_text16le(sqlite3_value*);
int sqlite3_value_type(sqlite3_value*);
```

This group of functions returns information about arguments passed to a user-defined function. Function implementations use these routines to access their arguments. These routines are the same as the `sqlite3_column()` routines except that these routines take a single `sqlite3_value` pointer instead of a `sqlite3_stmt` pointer along with an integer column number.

See the documentation under `sqlite3_column_blob()` for additional information.

sqlite3_vmprintf

```
char *sqlite3_vmprintf(const char*, va_list);
```

See `sqlite3_mprintf()`.

APPENDIX C

Codd's 12 Rules

The following rules are taken directly from Codd's 1985 article, "Is your DBMS really relational?" in *Computerworld* magazine. They describe the essential characteristics of the relational model. All of these rules are covered in detail in Chapter 3.

0. Rule Zero:

For any system that is advertised as, or claimed to be, a relational data base management system, that system must be able to manage data bases entirely through its relational capabilities.

1. The information rule:

All information in a relational data base is represented explicitly at the logical level and in exactly one way—by values in tables.

2. The guaranteed access rule:

Each and every datum (atomic value) in a relational data base is guaranteed to be logically accessible by resorting to a combination of table name, primary key value, and column name.

3. Systematic treatment of null values:

Null values (distinct from the empty character string or a string of blank characters and distinct from zero or any other number) are supported in fully relational DBMS for representing missing information and inapplicable information in a systematic way, independent of data type.

4. Dynamic online catalog based on the relational model:

The data base description is represented at the logical level in the same way as ordinary data, so that authorized users can apply the same relational language to its interrogation as they apply to the regular data.

5. The comprehensive data sublanguage rule:

A relational system may support several languages and various modes of terminal use (for example, the fill-in-the-blanks mode). However, there must be at least one language

whose statements are expressible, per some well-defined syntax, as character strings and that is comprehensive in supporting all the following items:

(a) Data Definition

(b) View Definition

(c) Data Manipulation (interactive and by program)

(d) Integrity Constraints, and Authorization

(e) Transaction boundaries (begin, commit, and rollback)

6. The view updating rule:

   All views that are theoretically updatable are also updatable by the system.

7. High-level insert, update, and delete:

   The capability of handling a base relation or a derived relation as a single operand applies not only to the retrieval of data but also to the insertion, update, and deletion of data.

8. Physical data independence:

   Application programs and terminal activities remain logically unimpaired whenever any changes are made in either storage representations or access methods.

9. Logical data independence:

   Application programs and terminal activities remain logically unimpaired when information-preserving changes of any kind that theoretically permit un-impairment are made to the base tables.

10. Integrity independence:

    Integrity constraints specific to a particular relational data base must be definable in the relational data sub-language and storable in the catalog, not in the application programs.

11. Distribution independence:

    A fully relational DBMS that does not support distributed data bases has the capability of being extended to provide that support while leaving application programs and terminal activities logically unimpaired, both at the time of initial distribution and whenever later redistribution is made.

12. The nonsubversion rule:

    If a relational system has a low-level (single-record-at-a-time) language, that low level cannot be used to subvert or bypass the integrity rules and constraints expressed in the higher level relational language (multiple-records-at-a-time).

Index

C

D

E

F

L

M

N

O

P

Q

R

S

T

U

V

W