SQL As Understood By SQLite

(This page was last modified on 2003/07/20 01:16:48 UTC)

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 document attempts to describe percisely what parts of the SQL language SQLite does and does not support. A list of keywords is given at the end.

In all of the syntax diagrams that follow, literal text is shown in bold blue. Non-terminal symbols are shown in italic red. Operators that are part of the syntactic markup itself are shown in black roman.

This document is just an overview of the SQL syntax implemented by SQLite. 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".

SQLite implements the follow syntax:

- ATTACH DATABASE
- BEGIN TRANSACTION
- comment
- COMMIT TRANSACTION
- COPY
- CREATE INDEX
- CREATE TABLE
- CREATE TRIGGER
- CREATE VIEW
- DELETE
- DETACH DATABASE
- DROP INDEX
- DROP TABLE
- DROP TRIGGER
- DROP VIEW
- END TRANSACTION
- EXPLAIN
- expression
- INSERT
- ON CONFLICT clause
- PRAGMA
- REPLACE
- ROLLBACK TRANSACTION
- SELECT
- UPDATE
- VACUUM

Details on the implementation of each command are provided in the sequel.

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, but you cannot alter the schema of an attached database. You can only CREATE and DROP in the original database.

You cannot create a new table with the same name as a table in an attached database, but you can attach a database which 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 which don't have duplicate names become the 'default' table of that name. Any tables of that name attached afterwards 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.

When there are attached databases, transactions are not atomic. Transactions continue to be atomic within each individual database file. But if your machine crashes in the middle of a COMMIT where you have updated two or more database files, some of those files might get the changes where others might not.

There is a compile-time limit of 10 attached database files.

Executing a BEGIN TRANSACTION statement locks all database files, so this feature cannot (currently) be used to increase concurrancy.

BEGIN TRANSACTION

sql-statement ::= **BEGIN** [**TRANSACTION** [*name*]] [**ON CONFLICT** *conflict-algorithm*]

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. See ATTACH for an exception when there are attached databases.

The optional transaction name is ignored. SQLite currently doesn't allow nested transactions. Attempting to start a new transaction inside another is an error.

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 documention on the ON CONFLICT clause for additional information about the ROLLBACK conflict resolution algorithm.

The optional ON CONFLICT clause at the end of a BEGIN statement can be used to changed the default conflict resolution algorithm. The normal default is ABORT. If an alternative is specified by the ON CONFLICT clause of a BEGIN, then that alternative is used as the default for all commands within the transaction. The default algorithm is overridden by ON CONFLICT clauses on individual constraints within the CREATE TABLE or CREATE INDEX statements and by the OR clauses on COPY, INSERT, and UPDATE commands.

---

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 whitespace by the parser. They can begin anywhere whitespace 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 whitespace 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.

COPY

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

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 which is to be filled with data. The filename is a string or 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 baskslash "\" character. A baskslash in the data appears as two backslashes in a row. The optional USING DELIMITERS clause can specify a delimiter other than tab.

If a column consists of the character "\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 for this one command. See the section titled ON CONFLICT for additional information.

When the input data source is STDIN, the input can be terminated by a line that contains only a baskslash and a dot: "\.".

CREATE INDEX

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

column-name ::= *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 one of the "ASC" or "DESC" keywords to indicate sort order, but the sort order is ignored in the current implementation. Sorting is always done in ascending order.

There are no arbitrary limits on the number of indices 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 not 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 titled 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. Everytime 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.

Non-temporary indexes cannot be added on tables in attached databases. They 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** *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*] | **UNIQUE** [*conflict-clause*] | **CHECK (** *expr* **)** [*conflict-clause*] | **DEFAULT** *value*

constraint ::= **PRIMARY KEY (** *name* [**,** *name*]* **)** [*conflict-clause*]| **UNIQUE (** *name* [**,** *name*]* **)** [*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. Tables names that begin with "**sqlite_**" are reserved for use by the engine.

Each column definition is the name of the column followed by the datatype for that column, then one or more optional column constraints. SQLite is typeless. The datatype for the column does not restrict what data may be put in that column. All information is stored as null-terminated strings. The UNIQUE constraint causes an index to be created on the specified

columns. This index must contain unique keys. The DEFAULT constraint specifies a default value to use when doing an INSERT.

Specifying a PRIMARY KEY normally just creates a UNIQUE index on the primary key. However, if primary key is on a single column that has datatype 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 for this one case, SQLite ignores the datatype specification of columns and allows any kind of data to be put in a column regardless of its declared datatype.) If a table does not have an INTEGER PRIMARY KEY column, then the B-Tree key will be a 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 or not there is an INTEGER PRIMARY KEY.

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 indices created on a temporary table are also temporary. Temporary tables and indices are stored in a separate file distinct from the main database file.

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 ABORT. Different constraints within the same table may have different default conflict resolution algorithms. If an 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 the section titled ON CONFLICT for additional information.

CHECK constraints are ignored in the current implementation. Support for CHECK constraints may be added in the future. As of version 2.3.0, NOT NULL, PRIMARY KEY, and UNIQUE constraints all work.

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 1 megabytes. (This limit can be increased to 16MB by changing a single #define in the source code and recompiling.)

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. Everytime 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 then an equivalent CREATE TABLE statement is synthesized and store in **sqlite_master** in place of the original command. The text of CREATE TEMPORARY TABLE statements are stored in the **sqlite_temp_master** table.

Tables are removed using the DROP TABLE statement. Non-temporary tables in an attached database cannot be dropped.

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-event*s 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).

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 of 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. Non-temporary triggers cannot be added on a table in an attached database.

CREATE VIEW

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

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

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. Non-temporary views cannot be created on tables in an attached database.

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** [*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.
Non-temporary indexes on tables in an attached database cannot be dropped.

DROP TABLE

sql-command ::= **DROP TABLE** *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 can not be recovered. All indices associated with the table are also deleted. Non-temporary tables in an attached database cannot be dropped.

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. Non-temporary triggers cannot be dropped on attached tables.

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. Non-temporary views in attached databases cannot be dropped.

EXPLAIN

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

The EXPLAIN command modifier is a non-standard 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 the architecture description or the documentation on available opcodes for the virtual machine.

---

expression

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

like-op ::= **LIKE** | **GLOB** | **NOT LIKE** | **NOT GLOB**

This section is different from the others. Most other sections of this document talks about a particular SQL command. This section does not talk about a standalone command but about "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 operaters are these:

```
-     +     !     ~
```

Any SQLite value can be used as part of an expression. For arithmetic operations, integers are treated as integers. Strings are first converted to real numbers using **atof()**. For comparison operators, numbers compare as numbers and strings compare using the **strcmp()** function. Note that there are two variations of the equals and not 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 LIKE operator does a wildcard comparision. The operand to the right contains the wildcards. A percent symbol % in the right operand matches any sequence of zero or more characters on the left. An underscore _ on the right matches any single character on the left. The LIKE operator is not case sensitive and will match upper case characters on one side against lower case characters on the other. (A bug: SQLite only understands upper/lower case 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.). The infix LIKE operator is identical the user function like(*X*,*Y*).

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 identical the user function glob(*X*,*Y*).

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.

SQLite supports a minimal Oracle8 outer join behavior. A column expression of the form "column" or "table.column" can be followed by the special "**(+)**" operator. If the table of the column expression is the second or subsequent table in a join, then that table becomes the left table in a LEFT OUTER JOIN. The expression that uses that table becomes part of the ON clause for the join. The exact Oracle8 behavior is not implemented, but it is possible to construct queries that will work correctly for both SQLite and Oracle8.

SELECT statements can appear in expressions as either the right-hand operand of the IN operator or as a scalar quantity. In both cases, the SELECT should have only a single column in its result. Compound SELECTs (connected with keywords like UNION or EXCEPT) are allowed. A SELECT in an expression is evaluated once before any other processing is performed, so none of the expressions within the select itself can refer to quantities in the containing expression.

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 yeilds no rows, then the value of the SELECT is NULL.

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 below are available by default. Additional functions may be written in C and added to the database engine using the sqlite_create_function() API.

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 2 arguments.

glob(*X*,*Y*) — This function is used to implement the "**Y GLOB X**" syntax of SQLite. The sqlite_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*) — This function is used to implement the "**Y LIKE X**" syntax of SQL. The sqlite_create_function() interface can be used to override this function and thereby change the operation of the LIKE operator.

lower(*X*) — Return a copy of string *X* will all characters converted to lower case. 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 2 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 mminimum value is determined by the usual sort order. Note that **min()** is a simple function when it has 2 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.

random(*) — Return a random integer between -2147483648 and +2147483647.

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 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 which is *Z* characters long. The left-most character of *X* is number 1. If *Y* is negative the 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 indices refer to actual UTF-8 characters, not bytes.

typeof(*X*) — Return the type of the expression *X*. The only return values are "numeric" and "text". SQLite's type handling is explained in Datatypes in SQLite.

upper(*X*) — Return a copy of input string *X* converted to all upper-case letters. The implementation of this function uses the C library routine **toupper()** which means it may not work correctly on UTF-8 strings.

The following aggregate functions are available by default. Additional aggregate functions written in C may be added using the sqlite_create_aggregate() API.

avg(*X*) — Return the average value of all *X* within a group.

count(*X*)
count(*) — The first form return a count of the number of times that *X* is not NULL in a group. The second form (with no argument) 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 value of all values in the group. The usual sort order is used to determine the minimum.

sum(*X*) — Return the numeric sum of all values in the group.

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 a 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 not default value is specified.

The second form of the INSERT statement takes it 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 columns name 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 titled 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 clause

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 non-standard clause that can appear in many other SQL commands. It is given its own section in this document because it is not part of standard SQL and therefore might not be familiar.

The syntax for the ON CONFLICT clause is as shown above for the CREATE TABLE, CREATE INDEX, and BEGIN TRANSACTION commands. For the COPY, INSERT, and UPDATE commands, the keywords "ON CONFLICT" are replaced by "OR", to make the syntax seem more natural. 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. This is what they mean:

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 continue to be inserted or updated normally. No error is returned.

REPLACE

When a UNIQUE constraint violation occurs, the pre-existing 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.

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

The conflict resolution algorithm can be specified in three places, in order from lowest to highest precedence:

1. On individual constraints within a CREATE TABLE or CREATE INDEX statement.
2. On a BEGIN TRANSACTION command.
3. In the OR clause of a COPY, INSERT, or UPDATE command.

The algorithm specified in the OR clause of a COPY, INSERT, or UPDATE overrides any algorithm specified on the BEGIN TRANSACTION command and the algorithm specified on the BEGIN TRANSACTION command overrides the algorithm specified in the a CREATE TABLE or CREATE INDEX. If no algorithm is specified anywhere, the ABORT algorithm is used.

PRAGMA

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

The PRAGMA command is used to modify the operation of the SQLite library. The pragma command is experimental and specific pragma statements may be removed or added in future releases of SQLite. Use this command with caution.

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 current implementation supports the following pragmas:

- **PRAGMA cache_size;**
  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 UPDATEs or DELETEs 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 count_changes = ON;** (1)

PRAGMA count_changes = OFF; (0)

When on, the COUNT_CHANGES pragma causes the callback function to be invoked once for each DELETE, INSERT, or UPDATE operation. The argument is the number of rows that were changed.

This pragma may be removed from future versions of SQLite. Consider using the **sqlite_changes()** API function instead.

- **PRAGMA database_list;**

  For each open database, invoke the callback function once with information about that database. Arguments include the index and the name the datbase 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 default_cache_size;**
  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 1K on disk and about 1.5K 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 everytime you reopen the database.

- **PRAGMA default_synchronous;**
  PRAGMA default_synchronous = FULL; (2)
  PRAGMA default_synchronous = NORMAL; (1)
  PRAGMA default_synchronous = OFF; (0)

  Query or change the setting of the "synchronous" flag in the database. 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 (1, the default), 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. So NORMAL is the default mode. 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.

  This pragma changes the synchronous mode persistently. Once changed, the mode stays as set even if the database is closed and reopened. The **synchronous** pragma does the same thing but only applies the setting to the current session.

- **PRAGMA default_temp_store;**
  PRAGMA default_temp_store = DEFAULT; (0)
  PRAGMA default_temp_store = MEMORY; (2)
  PRAGMA default_temp_store = FILE; (1)

  Query or change the setting of the "temp_store" flag stored in the database. When temp_store is DEFAULT (0), the compile-time default is used for the temporary database. When temp_store is MEMORY (2), an in-memory database is used. When temp_store is FILE (1), a temporary database file on disk will be used. Note that it is possible for the library compile-time options to override this setting. Once the temporary database is in

use, its location cannot be changed.

This pragma changes the temp_store mode persistently. Once changed, the mode stays as set even if the database is closed and reopened. The **temp_store** pragma does the same thing but only applies the setting to the current session.

- **PRAGMA empty_result_callbacks = ON;** (1)
  PRAGMA empty_result_callbacks = OFF; (0)

  When on, the EMPTY_RESULT_CALLBACKS pragma causes the callback function to be invoked once for each query that has an empty result set. The third "**argv**" parameter to the callback is set to NULL because there is no data to report. But the second "**argc**" and fourth "**columnNames**" parameters are valid and can be used to determine the number and names of the columns that would have been in the result set had the set not been empty.

- **PRAGMA full_column_names = ON;** (1)
  PRAGMA full_column_names = OFF; (0)

  The column names reported in an SQLite callback are normally just the name of the column itself, except for joins when "TABLE.COLUMN" is used. But when full_column_names is turned on, column names are always reported as "TABLE.COLUMN" even for simple queries.

- **PRAGMA index_info(***index-name***);**

  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***);**

  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 or not the index must be unique.

- **PRAGMA integrity_check;**

  The command does an integrity check of the entire database. It looks for out-of-order records, missing pages, malformed records, and corrupt indices. 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 = ON;** (1)
  PRAGMA parser_trace = OFF; (0)

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

- **PRAGMA show_datatypes = ON;** (1)
  PRAGMA show_datatypes = OFF; (0)

  When turned on, the SHOW_DATATYPES pragma causes extra entries containing the names of datatypes of columns to be appended to the 4th ("columnNames") argument to **sqlite_exec()** callbacks. When turned off, the 4th argument to callbacks contains only the column names. The datatype for table columns is taken from the CREATE TABLE statement that defines the table. Columns with an unspecified datatype have a datatype of "NUMERIC" and the results of expression have a datatype of either "TEXT" or "NUMERIC" depending on the expression. The following chart illustrates the difference for the query "SELECT 'xyzzy', 5, NULL AS empty ":

show_datatypes=OFF	**show_datatypes=ON**
azCol[0] = "xyzzy";	azCol[0] = "xyzzy";
azCol[1] = "5";	azCol[1] = "5";
azCol[2] = "empty";	azCol[2] = "empty";
azCol[3] = 0;	azCol[3] = "TEXT";
	azCol[4] =

```
"NUMERIC";
azCol[5] = "TEXT";
azCol[6] = 0;
```

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

  Query or change the setting of the "synchronous" flag affecting the database for the duration of the current database connection. The synchronous flag reverts to its default value when the database is closed and reopened. For additional information on the synchronous flag, see the description of the **default_synchronous** pragma.

- **PRAGMA table_info(***table-name***);**

  For each column in the named table, invoke the callback function once with information about that column, including the column name, data type, whether or not the column can be NULL, and the default value for the column.

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

  Query or change the setting of the "temp_store" flag affecting the database for the duration of the current database connection. The temp_store flag reverts to its default value when the database is closed and reopened. For additional information on the temp_store flag, see the description of the **default_temp_store** pragma. Note that it is possible for the library compile-time options to override this setting.

- **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.

No error message is generated if an unknown pragma is issued. Unknown pragmas are ignored.

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.

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-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 ::= **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 eachother. 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 tables 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 sub-query 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 clauses 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 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.

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 are 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 constraint conflict resolution algorithm to use during this one command. See the section titled ON CONFLICT for additional information.

VACUUM

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

The VACUUM command is an SQLite extension modelled after a similar command found in PostgreSQL. If VACUUM is invoked with the name of a table or index then it is suppose 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 reimplimented in version 2.8.1. It now cleans the database by copying its contents to a temporary database file and reloading the original database file from the copy. This will eliminate free pages, align table data to be contiguous, and otherwise clean up the database file structure. The index or table name argument is now ignored.

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

SQLite keywords

The following keywords are used by SQLite. Most are either reserved words in SQL-92 or were listed as potential reserved words. Those which aren't are shown in italics. Not all of these words are actually used by SQLite. Keywords are not reserved in SQLite. Any keyword can be used as an identifier for SQLite objects (columns, databases, indexes, tables, triggers, views, ...) but must generally be enclosed by brackets or quotes to avoid confusing the parser. Keyword matching in SQLite is case-insensitive.

Keywords can be used as identifiers in three ways:

'keyword'	Interpreted as a literal string if it occurs in a legal string context, otherwise as an identifier.
"keyword"	Interpreted as an identifier if it matches a known identifier and occurs in a legal identifier context, otherwise as a string.
[keyword]	Always interpreted as an identifer. (This notation is used by MS Access and SQL Server.)

Fallback Keywords

These keywords can be used as identifiers for SQLite objects without delimiters.

ABORT AFTER ASC *ATTACH* BEFORE BEGIN DEFERRED CASCADE *CLUSTER* *CONFLICT* *COPY* CROSS *DATABASE* *DELIMITERS* DESC *DETACH* EACH END *EXPLAIN* *FAIL* FOR FULL IGNORE IMMEDIATE INITIALLY INNER *INSTEAD* KEY LEFT MATCH NATURAL OF *OFFSET* OUTER *PRAGMA* *RAISE* *REPLACE* RESTRICT RIGHT *ROW* *STATEMENT* *TEMP* TEMPORARY TRIGGER *VACUUM* VIEW

Normal keywords

These keywords can be used as identifiers for SQLite objects, but must be enclosed in brackets or quotes for SQLite to recognize them as an identifier.

ALL AND AS BETWEEN BY CASE CHECK COLLATE COMMIT CONSTRAINT CREATE DEFAULT DEFERRABLE DELETE DISTINCT DROP ELSE EXCEPT FOREIGN FROM *GLOB* GROUP HAVING IN *INDEX* INSERT INTERSECT INTO IS *ISNULL* JOIN LIKE LIMIT NOT *NOTNULL* NULL ON OR ORDER PRIMARY REFERENCES ROLLBACK SELECT SET TABLE THEN TRANSACTION UNION UNIQUE UPDATE USING VALUES WHEN WHERE

Special words

The following are not keywords in SQLite, but are used as names of system objects. They can be used as an identifier for a different type of object.

*_ROWID_* *MAIN* OID *ROWID* *SQLITE_MASTER* *SQLITE_TEMP_MASTER*


Back to the SQLite Home Page