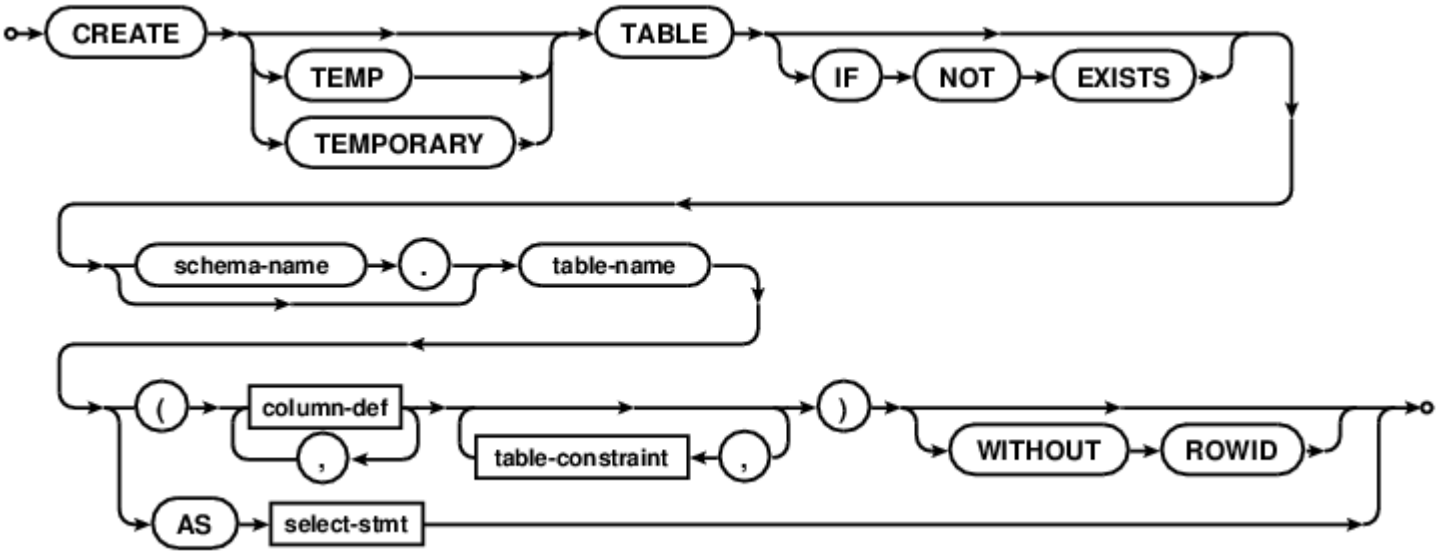
SQL As Understood By SQLite

[Top]

CREATE TABLE

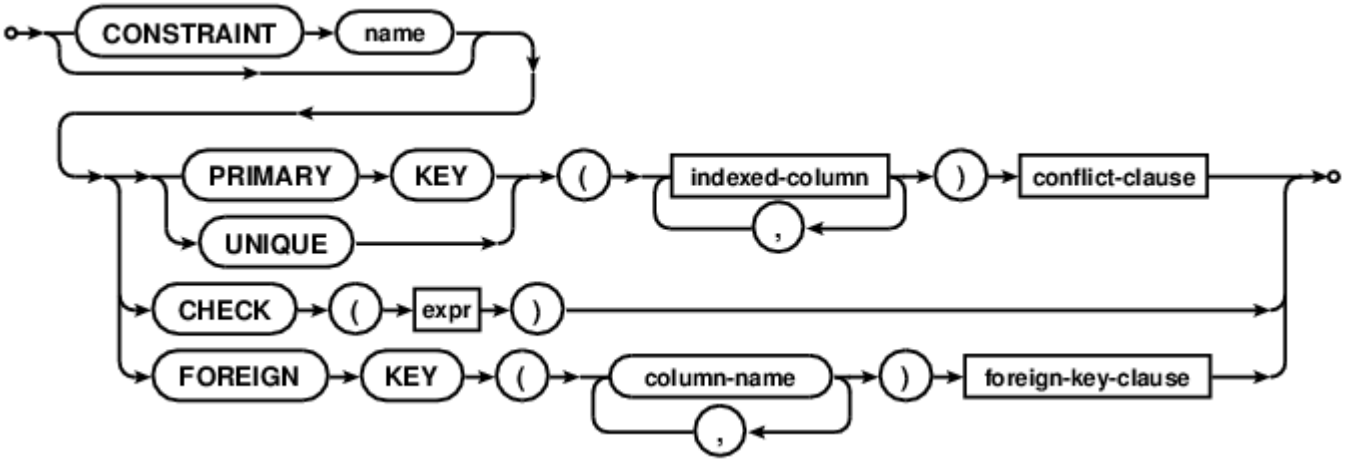
create-table-stmt: hide



column-def: show

select-stmt: show

table-constraint: hide



conflict-clause: show

expr: show

foreign-key-clause: show

indexed-column: show

The "CREATE TABLE" command is used to create a new table in an SQLite database. A CREATE TABLE command specifies the following attributes of the new table:

- The name of the new table.
- The database in which the new table is created. Tables may be created in the main database, the temp database, or in any attached database.
- The name of each column in the table.
- The declared type of each column in the table.
- A default value or expression for each column in the table.
- A default collation sequence to use with each column.
- Optionally, a PRIMARY KEY for the table. Both single column and composite (multiple column) primary keys are supported.
- A set of SQL constraints for each table. SQLite supports UNIQUE, NOT NULL, CHECK and FOREIGN KEY constraints.
- Whether the table is a WITHOUT ROWID table.

Every CREATE TABLE statement must specify a name for the new table. Table names that begin with "sqlite_" are reserved for internal use. It is an error to attempt to create a table with a name that starts with "sqlite_".

If a (schema-name) is specified, it must be either "main", "temp", or the name of an attached database. In this case the new table is created in the named database. If the "TEMP" or "TEMPORARY" keyword occurs between the "CREATE" and "TABLE" then the

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

It is usually an error to attempt to create a new table in a database that already contains a table, index or view of the same name. However, if the "IF NOT EXISTS" clause is specified as part of the CREATE TABLE statement and a table or view of the same name already exists, the CREATE TABLE command simply has no effect (and no error message is returned). An error is still returned if the table cannot be created because of an existing index, even if the "IF NOT EXISTS" clause is specified.

It is not an error to create a table that has the same name as an existing trigger.

Tables are removed using the DROP TABLE statement.

CREATE TABLE ... AS SELECT Statements

A "CREATE TABLE ... AS SELECT" statement creates and populates a database table based on the results of a SELECT statement. The table has the same number of columns as the rows returned by the SELECT statement. The name of each column is the same as the name of the corresponding column in the result set of the SELECT statement. The declared type of each column is determined by the expression affinity of the corresponding expression in the result set of the SELECT statement, as follows:

Expression Affinity	Column Declared Type
TEXT	"TEXT"
NUMERIC	"NUM"
INTEGER	"INT"
REAL	"REAL"
NONE	"" (empty string)

A table created using CREATE TABLE AS has no PRIMARY KEY and no constraints of any kind. The default value of each column is NULL. The default collation sequence for each column of the new table is BINARY.

Tables created using CREATE TABLE AS are initially populated with the rows of data returned by the SELECT statement. Rows are assigned contiguously ascending rowid values, starting with 1, in the order that they are returned by the SELECT statement.

Column Definitions

Unless it is a CREATE TABLE ... AS SELECT statement, a CREATE TABLE includes one or more column definitions, optionally followed by a list of table constraints. Each column definition consists of the name of the column, optionally followed by the declared type of the column, then one or more optional column constraints. Included in the definition of "column constraints" for the purposes of the previous statement are the COLLATE and DEFAULT clauses, even though these are not really constraints in the sense that they do not restrict the data that the table may contain. The other constraints - NOT NULL,

CHECK, UNIQUE, PRIMARY KEY and FOREIGN KEY constraints - impose restrictions on the tables data, and are are described under SQL Data Constraints below.

Unlike most SQL databases, SQLite does not restrict the type of data that may be inserted into a column based on the columns declared type. Instead, SQLite uses dynamic typing. The declared type of a column is used to determine the affinity of the column only.

The DEFAULT clause specifies a default value to use for the column if no value is explicitly provided by the user when doing an INSERT. If there is no explicit DEFAULT clause attached to a column definition, then the default value of the column is NULL. An explicit DEFAULT clause may specify that the default value is NULL, a string constant, a blob constant, a signed-number, or any constant expression enclosed in parentheses. A default value may also be one of the special case-independent keywords CURRENT_TIME, CURRENT_DATE or CURRENT_TIMESTAMP. For the purposes of the DEFAULT clause, an expression is considered constant if it does contains no sub-queries, column or table references, bound parameters, or string literals enclosed in double-quotes instead of single-quotes.

Each time a row is inserted into the table by an INSERT statement that does not provide explicit values for all table columns the values stored in the new row are determined by their default values, as follows:

- If the default value of the column is a constant NULL, text, blob or signed-number value, then that value is used directly in the new row.
- If the default value of a column is an expression in parentheses, then the expression is evaluated once for each row inserted and the results used in the new row.
- If the default value of a column is CURRENT_TIME, CURRENT_DATE or CURRENT_TIMESTAMP, then the value used in the new row is a text representation of the current UTC date and/or time. For CURRENT_TIME, the format of the value is "HH:MM:SS". For CURRENT_DATE, "YYYY-MM-DD". The format for CURRENT_TIMESTAMP is "YYYY-MM-DD HH:MM:SS".

The COLLATE clause specifies the name of a collating sequence to use as the default collation sequence for the column. If no COLLATE clause is specified, the default collation sequence is BINARY.

The number of columns in a table is limited by the SQLITE_MAX_COLUMN compile-time parameter. A single row of a table cannot store more than SQLITE_MAX_LENGTH bytes of data. Both of these limits can be lowered at runtime using the sqlite3_limit() C/C++ interface.

SQL Data Constraints

Each table in SQLite may have at most one **PRIMARY KEY**. If the keywords PRIMARY KEY are added to a column definition, then the primary key for the table consists of that single column. Or, if a PRIMARY KEY clause is specified as a table-constraint, then the primary key of the table consists of the list of columns specified as part of the PRIMARY KEY clause. The PRIMARY KEY clause must contain only column names — the use of

expressions in an indexed-column of a PRIMARY KEY is not supported. An error is raised if more than one PRIMARY KEY clause appears in a CREATE TABLE statement. The PRIMARY KEY is optional for ordinary tables but is required for WITHOUT ROWID tables.

If a table has a single column primary key and the declared type of that column is "INTEGER" and the table is not a WITHOUT ROWID table, then the column is known as an INTEGER PRIMARY KEY. See below for a description of the special properties and behaviors associated with an INTEGER PRIMARY KEY.

Each row in a table with a primary key must have a unique combination of values in its primary key columns. For the purposes of determining the uniqueness of primary key values, NULL values are considered distinct from all other values, including other NULLs. If an INSERT or UPDATE statement attempts to modify the table content so that two or more rows have identical primary key values, that is a constraint violation.

According to the SQL standard, PRIMARY KEY should always imply NOT NULL. Unfortunately, due to a bug in some early versions, this is not the case in SQLite. Unless the column is an INTEGER PRIMARY KEY or the table is a WITHOUT ROWID table or the column is declared NOT NULL, SQLite allows NULL values in a PRIMARY KEY column. SQLite could be fixed to conform to the standard, but doing so might break legacy applications. Hence, it has been decided to merely document the fact that SQLite allowing NULLs in most PRIMARY KEY columns.

A **UNIQUE** constraint is similar to a PRIMARY KEY constraint, except that a single table may have any number of UNIQUE constraints. For each UNIQUE constraint on the table, each row must contain a unique combination of values in the columns identified by the UNIQUE constraint. For the purposes of UNIQUE constraints, NULL values are considered distinct from all other values, including other NULLs. As with PRIMARY KEYs, a UNIQUE table-constraint clause must contain only column names — the use of expressions in an indexed-column of a UNIQUE table-constraint is not supported.

In most cases, UNIQUE and PRIMARY KEY constraints are implemented by creating a unique index in the database. (The exceptions are INTEGER PRIMARY KEY and PRIMARY KEYs on WITHOUT ROWID tables.) Hence, the following schemas are logically equivalent:

1. CREATE TABLE t1(a, b UNIQUE);
2. CREATE TABLE t1(a, b PRIMARY KEY);
3. CREATE TABLE t1(a, b);
   CREATE UNIQUE INDEX t1b ON t1(b);

A **CHECK** constraint may be attached to a column definition or specified as a table constraint. In practice it makes no difference. Each time a new row is inserted into the table or an existing row is updated, the expression associated with each CHECK constraint is evaluated and cast to a NUMERIC value in the same way as a CAST expression. If the result is zero (integer value 0 or real value 0.0), then a constraint violation has occurred. If the CHECK expression evaluates to NULL, or any other non-zero value, it is not a constraint violation. The expression of a CHECK constraint may not contain a subquery.

A **NOT NULL** constraint may only be attached to a column definition, not specified as a table constraint. Not surprisingly, a NOT NULL constraint dictates that the associated

column may not contain a NULL value. Attempting to set the column value to NULL when inserting a new row or updating an existing one causes a constraint violation.

Exactly how a constraint violation is dealt with is determined by the constraint conflict resolution algorithm. Each PRIMARY KEY, UNIQUE, NOT NULL and CHECK constraint has a default conflict resolution algorithm. PRIMARY KEY, UNIQUE and NOT NULL constraints may be explicitly assigned a default conflict resolution algorithm by including a conflict-clause in their definitions. Or, if a constraint definition does not include a conflict-clause or it is a CHECK constraint, the default conflict resolution algorithm is ABORT. Different constraints within the same table may have different default conflict resolution algorithms. See the section titled ON CONFLICT for additional information.

ROWIDs and the INTEGER PRIMARY KEY

Except for WITHOUT ROWID tables, all rows within SQLite tables have a 64-bit signed integer key that uniquely identifies the row within its table. This integer is usually called the "rowid". The rowid value can be accessed using one of the special case-independent names "rowid", "oid", or "_rowid_" in place of a column name. If a table contains a user defined column named "rowid", "oid" or "_rowid_", then that name always refers the explicitly declared column and cannot be used to retrieve the integer rowid value.

The rowid (and "oid" and "_rowid_") is omitted in WITHOUT ROWID tables. WITHOUT ROWID tables are only available in SQLite version 3.8.2 (2013-12-06) and later. A table that lacks the WITHOUT ROWID clause is called a "rowid table".

The data for rowid tables is stored as a B-Tree structure containing one entry for each table row, using the rowid value as the key. This means that retrieving or sorting records by rowid is fast. Searching for a record with a specific rowid, or for all records with rowids within a specified range is around twice as fast as a similar search made by specifying any other PRIMARY KEY or indexed value.

With one exception noted below, if a rowid table has a primary key that consists of a single column and the declared type of that column is "INTEGER" in any mixture of upper and lower case, then the column becomes an alias for the rowid. Such a column is usually referred to as an "integer primary key". A PRIMARY KEY column only becomes an integer primary key if the declared type name is exactly "INTEGER". Other integer type names like "INT" or "BIGINT" or "SHORT INTEGER" or "UNSIGNED INTEGER" causes the primary key column to behave as an ordinary table column with integer affinity and a unique index, not as an alias for the rowid.

The exception mentioned above is that if the declaration of a column with declared type "INTEGER" includes an "PRIMARY KEY DESC" clause, it does not become an alias for the rowid and is not classified as an integer primary key. This quirk is not by design. It is due to a bug in early versions of SQLite. But fixing the bug could result in backwards incompatibilities. Hence, the original behavior has been retained (and documented) because odd behavior in a corner case is far better than a compatibility break. This means that the following three table declarations all cause the column "x" to be an alias for the rowid (an integer primary key):

- `CREATE TABLE t(x INTEGER PRIMARY KEY ASC, y, z);`
- `CREATE TABLE t(x INTEGER, y, z, PRIMARY KEY(x ASC));`
- `CREATE TABLE t(x INTEGER, y, z, PRIMARY KEY(x DESC));`

But the following declaration does not result in "x" being an alias for the rowid:

- `CREATE TABLE t(x INTEGER PRIMARY KEY DESC, y, z);`

Rowid values may be modified using an UPDATE statement in the same way as any other column value can, either using one of the built-in aliases ("rowid", "oid" or "_rowid_") or by using an alias created by an integer primary key. Similarly, an INSERT statement may provide a value to use as the rowid for each row inserted. Unlike normal SQLite columns, an integer primary key or rowid column must contain integer values. Integer primary key or rowid columns are not able to hold floating point values, strings, BLOBs, or NULLs.

If an UPDATE statement attempts to set an integer primary key or rowid column to a NULL or blob value, or to a string or real value that cannot be losslessly converted to an integer, a "datatype mismatch" error occurs and the statement is aborted. If an INSERT statement attempts to insert a blob value, or a string or real value that cannot be losslessly converted to an integer into an integer primary key or rowid column, a "datatype mismatch" error occurs and the statement is aborted.

If an INSERT statement attempts to insert a NULL value into a rowid or integer primary key column, the system chooses an integer value to use as the rowid automatically. A detailed description of how this is done is provided separately.

The parent key of a foreign key constraint is not allowed to use the rowid. The parent key must used named columns only.