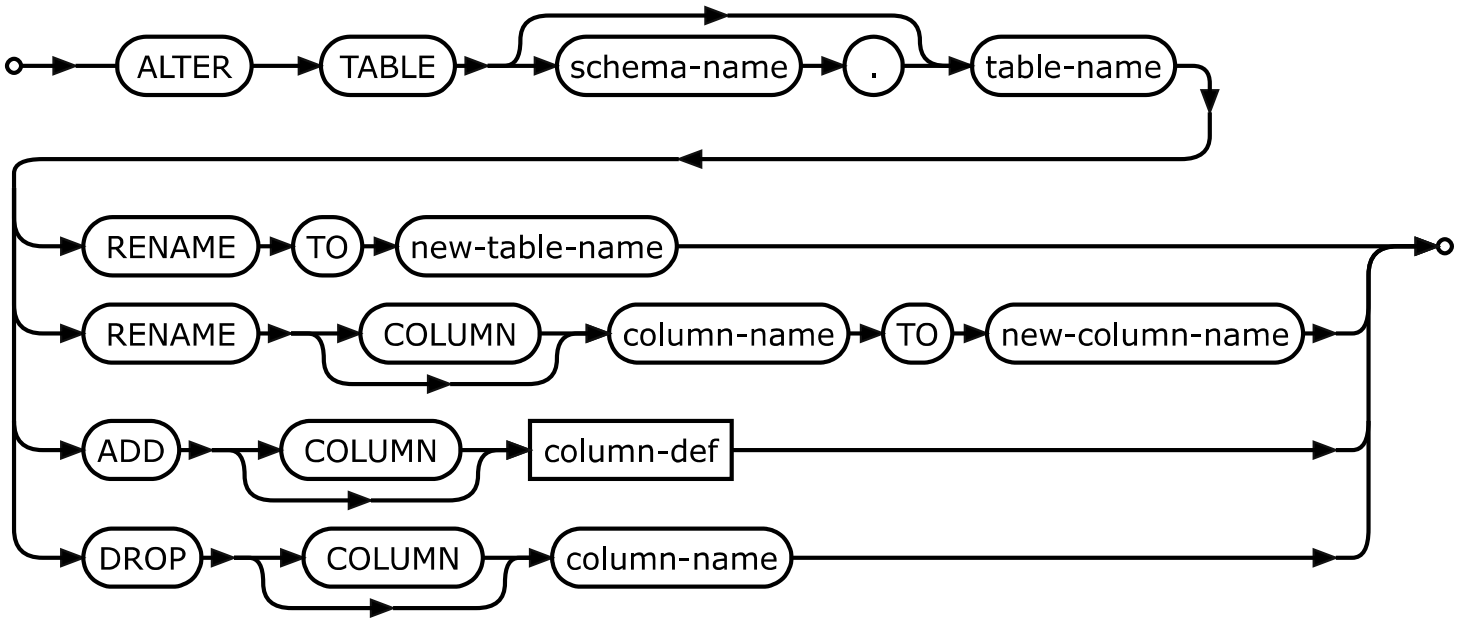
ALTER TABLE

1. Overview

alter-table-stmt: hide



column-def: show

SQLite supports a limited subset of ALTER TABLE. The ALTER TABLE command in SQLite allows these alterations of an existing table: it can be renamed; a column can be renamed; a column can be added to it; or a column can be dropped from it.

2. ALTER TABLE RENAME

The RENAME TO syntax changes the name of (table-name) to (new-table-name). This command cannot be used to move a table between attached databases, only to rename a table within the same database. If the table being renamed has triggers or indices, then these remain attached to the table after it has been renamed.

> **Compatibility Note:** The behavior of ALTER TABLE when renaming a table was enhanced in versions 3.25.0 (2018-09-15) and 3.26.0 (2018-12-01) in order to

carry the rename operation forward into triggers and views that reference the renamed table. This is considered an improvement. Applications that depend on the older (and arguably buggy) behavior can use the PRAGMA legacy_alter_table=ON statement or the SQLITE_DBCONFIG_LEGACY_ALTER_TABLE configuration parameter on sqlite3_db_config() interface to make ALTER TABLE RENAME behave as it did prior to version 3.25.0.

Beginning with release 3.25.0 (2018-09-15), references to the table within trigger bodies and view definitions are also renamed.

Prior to version 3.26.0 (2018-12-01), FOREIGN KEY references to a table that is renamed were only edited with PRAGMA foreign_keys=ON, or in other words if foreign key constraints were being enforced. With PRAGMA foreign_keys=OFF, FOREIGN KEY constraints would not be changed when the table that the foreign key referred to (the "parent table") was renamed. Beginning with version 3.26.0, FOREIGN KEY constraints are always converted when a table is renamed, unless the PRAGMA legacy_alter_table=ON setting is engaged. The following table summarizes the difference:

PRAGMA foreign_keys	**PRAGMA legacy_alter_table**	**Parent Table references are updated**	**SQLite version**
Off	Off	No	< 3.26.0
Off	Off	Yes	>= 3.26.0
On	Off	Yes	all
Off	On	No	all
On	On	Yes	all

3. ALTER TABLE RENAME COLUMN

The RENAME COLUMN TO syntax changes the column-name of table table-name into new-column-name. The column name is changed both within the table definition itself and also within all indexes, triggers, and views that reference the column. If the column name change would result in a semantic ambiguity in a trigger or view, then the RENAME COLUMN fails with an error and no changes are applied.

4. ALTER TABLE ADD COLUMN

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. The column-def rule defines the characteristics of the new column. The new column 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, CURRENT_TIMESTAMP, or an expression in parentheses.
- If a NOT NULL constraint is specified, then the column must have a default value other than NULL.
- If foreign key constraints are enabled and a column with a REFERENCES clause is added, the column must have a default value of NULL.
- The column may not be GENERATED ALWAYS ... STORED, though VIRTUAL columns are allowed.

When adding a column with a CHECK constraint, or a NOT NULL constraint on a generated column, the added constraints are tested against all preexisting rows in the table and the ADD COLUMN fails if any constraint fails. The testing of added constraints against preexisting rows is a new enhancement as of SQLite version 3.37.0 (2021-11-27).

The ALTER TABLE command works by modifying the SQL text of the schema stored in the sqlite_schema table. No changes are made to table content for renames or column addition without constraints. Because of this, the execution time of such ALTER TABLE commands is independent of the amount of data in the table and such commands will run as quickly on a table with 10 million rows as on a table with 1 row. When adding new columns that have CHECK constraints, or adding generated columns with NOT NULL constraints, or when deleting columns, then all existing data in the table must be either read (to test new constraints against existing rows) or written (to remove deleted columns). In those cases, the ALTER TABLE command takes time that is proportional to the amount of content in the table being altered.

After ADD COLUMN has been run on a database, that database will not be readable by SQLite version 3.1.3 (2005-02-20) and earlier.

5. ALTER TABLE DROP COLUMN

The DROP COLUMN syntax is used to remove an existing column from a table. The DROP COLUMN command removes the named column from the table, and rewrites its content to purge the data associated with that column. The DROP COLUMN command only works if the column is not referenced by any other parts of the schema and is not a PRIMARY KEY and does not have a UNIQUE constraint. Possible reasons why the DROP COLUMN command can fail include:

- The column is a PRIMARY KEY or part of one.
- The column has a UNIQUE constraint.
- The column is indexed.
- The column is named in the WHERE clause of a partial index.
- The column is named in a table or column CHECK constraint not associated with the column being dropped.
- The column is used in a foreign key constraint.
- The column is used in the expression of a generated column.
- The column appears in a trigger or view.

5.1. How It Works

SQLite stores the schema as plain text in the sqlite_schema table. All ALTER TABLE commands modify that text and then attempt to reparse the entire schema. The command is only successful if the schema is still valid after the text has been modified. In the case of the DROP COLUMN command, the only text modified is that the column definition is removed from the CREATE TABLE statement. The DROP COLUMN command will fail if there are any traces of the column in other parts of the schema that will prevent the schema from parsing after the CREATE TABLE statement has been modified.

6. Disable Error Checking Using PRAGMA writable_schema=ON

ALTER TABLE will normally fail and make no changes if it encounters any entries in the sqlite_schema table that do not parse. For example, if there is a malformed VIEW or TRIGGER associated with table named "tbl1", then an attempt to rename "tbl1" to "tbl1neo" will fail because the associated views and triggers could not be parsed.

Beginning with SQLite 3.38.0 (2022-02-22), this error checking can be disabled by setting "PRAGMA writable_schema=ON;". When the schema is writable, ALTER TABLE silently ignores any rows of the sqlite_schema table that do not parse.

7. Making Other Kinds Of Table Schema Changes

The only schema altering commands directly supported by SQLite are the "rename table", "rename column", "add column", "drop column" commands shown above. However, applications can make other arbitrary changes to the format of a table using a simple sequence of operations. The steps to make arbitrary changes to the schema design of some table X are as follows:

1. If foreign key constraints are enabled, disable them using PRAGMA foreign_keys=OFF.
2. Start a transaction.
3. Remember the format of all indexes, triggers, and views associated with table X. This information will be needed in step 8 below. One way to do this is to run a query like the following: SELECT type, sql FROM sqlite_schema WHERE tbl_name='X'.
4. Use CREATE TABLE to construct a new table "new_X" that is in the desired revised format of table X. Make sure that the name "new_X" does not collide with any existing table name, of course.
5. Transfer content from X into new_X using a statement like: INSERT INTO new_X SELECT ... FROM X.
6. Drop the old table X: DROP TABLE X.
7. Change the name of new_X to X using: ALTER TABLE new_X RENAME TO X.

8. Use CREATE INDEX, CREATE TRIGGER, and CREATE VIEW to reconstruct indexes, triggers, and views associated with table X. Perhaps use the old format of the triggers, indexes, and views saved from step 3 above as a guide, making changes as appropriate for the alteration.

9. If any views refer to table X in a way that is affected by the schema change, then drop those views using DROP VIEW and recreate them with whatever changes are necessary to accommodate the schema change using CREATE VIEW.

10. If foreign key constraints were originally enabled then run PRAGMA foreign_key_check to verify that the schema change did not break any foreign key constraints.

11. Commit the transaction started in step 2.

12. If foreign keys constraints were originally enabled, reenable them now.

Caution: Take care to follow the procedure above precisely. The boxes below summarize two procedures for modifying a table definition. At first glance, they both appear to accomplish the same thing. However, the procedure on the right does not always work, especially with the enhanced rename table capabilities added by versions 3.25.0 and 3.26.0. In the procedure on the right, the initial rename of the table to a temporary name might corrupt references to that table in triggers, views, and foreign key constraints. The safe procedure on the left constructs the revised table definition using a new temporary name, then renames the table into its final name, which does not break links.

1. Create new table 2. Copy data 3. Drop old table 4. Rename new into old	1. Rename old table 2. Create new table 3. Copy data 4. Drop old table
↑ **Correct**	↑ **Incorrect**

The 12-step generalized ALTER TABLE procedure above will work even if the schema change causes the information stored in the table to change. So the full 12-step procedure above is appropriate for dropping a column, changing the order of columns, adding or removing a UNIQUE constraint or PRIMARY KEY, adding CHECK or FOREIGN KEY or NOT NULL constraints, or changing the datatype for a column, for example. However, a simpler and faster procedure can optionally be used for some changes that do not affect the on-disk content in any way. The following simpler procedure is appropriate for removing CHECK or FOREIGN KEY or NOT NULL constraints, or adding, removing, or changing default values on a column.

1. Start a transaction.

2. Run PRAGMA schema_version to determine the current schema version number. This number will be needed for step 6 below.

3. Activate schema editing using PRAGMA writable_schema=ON.

4. Run an UPDATE statement to change the definition of table X in the sqlite_schema table: UPDATE sqlite_schema SET sql=... WHERE type='table' AND name='X';

   Caution: Making a change to the sqlite_schema table like this will render the database corrupt and unreadable if the change contains a syntax error. It is suggested that careful testing of the UPDATE statement be done on a separate blank database prior to using it on a database containing important data.

5. If the change to table X also affects other tables or indexes or triggers are views within schema, then run UPDATE statements to modify those other tables indexes and views too. For example, if the name of a column changes, all FOREIGN KEY constraints, triggers, indexes, and views that refer to that column must be modified.

   Caution: Once again, making changes to the sqlite_schema table like this will render the database corrupt and unreadable if the change contains an error. Carefully test this entire procedure on a separate test database prior to using it on a database containing important data and/or make backup copies of important databases prior to running this procedure.

6. Increment the schema version number using PRAGMA schema_version=X where X is one more than the old schema version number found in step 2 above.

7. Disable schema editing using PRAGMA writable_schema=OFF.

8. (Optional) Run PRAGMA integrity_check to verify that the schema changes did not damage the database.

9. Commit the transaction started on step 1 above.

If some future version of SQLite adds new ALTER TABLE capabilities, those capabilities will very likely use one of the two procedures outlined above.

8. Why ALTER TABLE is such a problem for SQLite

Most SQL database engines store the schema already parsed into various system tables. On those database engines, ALTER TABLE merely has to make modifications to the corresponding system tables.

SQLite is different in that it stores the schema in the sqlite_schema table as the original text of the CREATE statements that define the schema. Hence ALTER TABLE needs to revise the text of the CREATE statement. Doing so can be tricky for certain "creative" schema designs.

The SQLite approach of storing the schema as text has advantages for an embedded relational database. For one, it means that the schema takes up less space in the database file. This is important since a common SQLite usage pattern is to have many small, separate database files instead of putting everything in one big global database file, which is the usual approach for client/server database engines. Since the schema is

duplicated in each separate database file, it is important to keep the schema representation compact.

Storing the schema as text rather than as parsed tables also gives flexibility to the implementation. Since the internal parse of the schema is regenerated each time the database is opened, the internal representation of the schema can change from one release to the next. This is important, as sometimes new features require enhancements to the internal schema representation. Changing the internal schema representation would be much more difficult if the schema representation was exposed in the database file. So, in other words, storing the schema as text helps maintain backwards compatibility, and helps ensure that older database files can be read and written by newer versions of SQLite.

Storing the schema as text also makes the SQLite database file format easier to define, document, and understand. This helps make SQLite database files a recommended storage format for long-term archiving of data.

The downside of storing schema as text is that it can make the schema tricky to modify. And for that reason, the ALTER TABLE support in SQLite has traditionally lagged behind other SQL database engines that store their schemas as parsed system tables that are easier to modify.

This page last modified on 2025-09-19 22:09:24 UTC