Using

SQLite

O'REILLY®

Jay A. Kreibich

Table of Contents

APPENDIX C
SQLite SQL Command Reference

This appendix lists the SQL commands and syntax that are supported by SQLite. SQL statements consist of a single command and any required parameters. Command statements are separated by a semicolon. Technically, standalone statements do not need to be terminated with a semicolon, but most interactive environments require the use of a semicolon to indicate that the current command statement is complete and should be executed. For example, the C API `sqlite3_exec()` does not require that command statements end with a semicolon, but interactive use of `sqlite3` requires ending each statement with a semicolon.

In most situations where a table name is called for, a view name can be used instead. As noted in the syntax diagrams, in most instances where any object identifier is used (table name, view name, etc.), the name can be qualified with a logical database name to prevent any ambiguity between objects in different databases that share a similar name (see `ATTACH DATABASE` in this appendix). If the object is unqualified, it will be searched for in the `temp` database, followed by the `main` database, followed by each attached database, in order. If an unqualified identifier appears in a `CREATE` statement, the object will be created in the main database, unless the statement contains some type of `CREATE TEMPORARY` syntax. Object identifiers that use nonstandard characters must be quoted. See "Basic Syntax" on page 30 for more info.

The `SELECT`, `UPDATE`, and `DELETE` commands contain clauses that are used to define search criteria on table rows. These table references can include the nonstandard phrases `INDEXED BY` or `NOT INDEXED`, to indicate whether the query optimizer should (or should not) attempt to use an index to satisfy the search condition. These extensions are included in SQLite to assist with testing, debugging, and hand-tuning queries. Their use in production code is not recommended, and therefore they are not included in the syntax diagrams or command explanations found in this appendix. For more information, see the SQLite website (*http://www.sqlite.org/lang_indexedby.html*).

Appendix C. SQLite SQL Command Reference. Using SQLite, ISBN: 9781449394592

Finally, be aware that the syntax diagrams presented with each command should not be taken as the definitive specification for the full command syntax. Some rarely used, nonstandard syntax (such as the `INDEXED BY` extension discussed in the previous paragraph) are not included in these diagrams. Similarly, there are possible syntax combinations that the diagrams will indicate are possible, but do not actually form logical statements. For example, according to the syntax diagrams, a `JOIN` operator can contain both a prefixed `NATURAL` condition, as well as a trailing `ON` or `USING` condition. This isn't possible in practice, as a join is limited to only one type of condition. While it would have been possible to present the diagram with only the allowed syntax, the diagram would have become much larger and much more complex. In such situations, it was decided that making the diagram easy to understand was more important than making it walk an absolute line on what was allowed or not allowed. Thankfully, such situations are reasonably rare. Just don't assume that because the parser can parse it means that the command makes sense to the database engine.

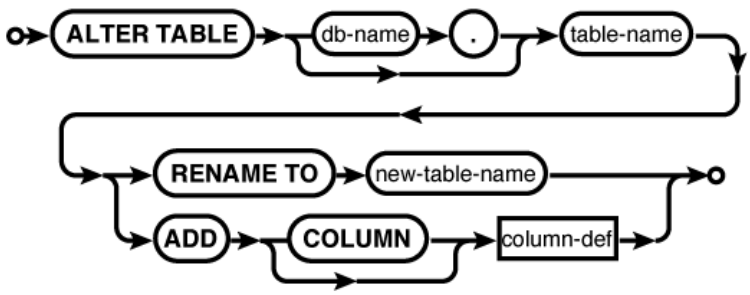
SQLite SQL Commands

The following SQL commands and syntax are supported by SQLite.

ALTER TABLE

Modify an existing table

Syntax



Common Usage

```
ALTER TABLE database_name.table_name RENAME TO new_table_name;
ALTER TABLE database_name.table_name ADD COLUMN column_def...;
```

Description

The `ALTER TABLE` command modifies an existing table without performing a full dump and reload of the data. The SQLite version of `ALTER TABLE` supports two basic operations. The `RENAME` variant is used to change the name of a table, while `ADD COLUMN` is used to add a new column to an existing table. Both versions of the `ALTER TABLE` command will retain any existing data in the table.

Appendix C. SQLite SQL Command Reference. Using SQLite, ISBN: 9781449394592

RENAME

The `RENAME` variant is used to "move" or rename an existing table. An `ALTER TABLE...RENAME` command can only modify a table in place, it cannot be used to move a table to another database. A database name can be provided when specifying the original table name, but only the table name should be given when specifying the new table name.

Indexes and triggers associated with the table will remain with the table under the new name. If foreign key support is enabled, any foreign keys that reference this table will also be updated.

View definitions and trigger statements that reference the table by name will *not* be modified. These statements must be dropped and recreated, or a replacement table must be created.

ADD COLUMN

The `ADD COLUMN` variant is used to add a new column to the end of a table definition. New columns must always go at the end of the table definition. The existing table rows are not actually modified, meaning that the added columns are implied until a row is modified. This means the `ALTER TABLE...ADD COLUMN` command is quite fast, even for large tables, but it also means there are some limitations on the columns that can be added.

The added column:

- Cannot have a `PRIMARY KEY` constraint
- Cannot have a `UNIQUE` constraint
- Must have a literal, non-NULL default value if a `NOT NULL` constraint is given
- Cannot have a default of `CURRENT_TIME`, `CURRENT_DATE`, or `CURRENT_TIMESTAMP`

If foreign key constraints are enabled and the added column is defined as a foreign key (it has a `REFERENCES` clause), the new column must have a default of NULL.

Additionally, if the new column has a `CHECK` constraint, that constraint will only be applied to new values. This can lead to data that is inconsistent with the `CHECK`.

There is no way to remove a column once it has been added.

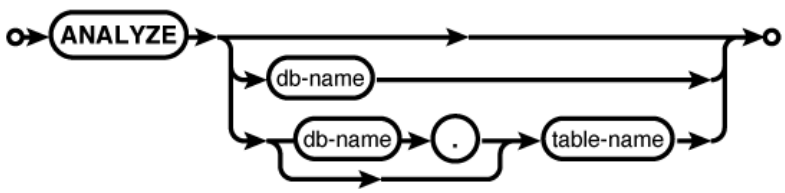
See Also

`CREATE TABLE`

ANALYZE

Compute index meta-data

Syntax



Appendix C. SQLite SQL Command Reference. Using SQLite, ISBN: 9781449394592

Common Usage

```
ANALYZE;
ANALYZE database_name;
ANALYZE database_name.table_name;
```

Description

The ANALYZE command computes and records statistical data about database indexes. If available, this data is used by the query optimizer to compute the most efficient query plan.

If no parameters are given, statistics will be computed for all indexes in all attached databases. You can also limit analysis to just those indexes in a specific database, or just those indexes associated with a specific table.

The statistical data is *not* automatically updated as the index values change. If the contents or distribution of an index changes significantly, it would be wise to reanalyze the appropriate database or table. Another option would be to simply delete the statistical data, as no data is usually better than incorrect data.

Data generated by ANALYZE is stored in one or more tables named sqlite_stat#, starting with sqlite_stat1. These tables cannot be manually dropped, but the data inside can be altered with standard SQL commands. Generally, this is not recommended, except to delete any ANALYZE data that is no longer valid or desired.

By default, the ANALYZE command generates data on the number of entries in an index, as well as the ratio of unique values to total values. This ratio is computed by dividing the total number of entries by the number of unique values, rounding up to the nearest integer. This data is used to compute the cost difference between a full-table scan and an indexed lookup.

If SQLite is compiled with the SQLITE_ENABLE_STAT2 directive, then ANALYZE will also generate an sqlite_stat2 table that contains a histogram of the index distribution. This is used to compute the cost of targeting a range of values.

There is one known issue with the ANALYZE command. When generating the sqlite_stat1 table, ANALYZE must calculate the number of unique values in an index. To accomplish this, the ANALYZE command uses the standard SQL test for equivalence between index values. This means that if a single-column index contains multiple NULL entries, they will each be considered a nonequivalent, unique value (since NULL != NULL). As a result, if an index contains a large number of NULL values, the ANALYZE data will incorrectly consider the index to have more uniqueness than it actually does. This can incorrectly influence the optimizer to pick an index-based lookup when a full-table scan would be less expensive. Due to this behavior, if an index contains a noticeable percentage of NULL entries (say, 10 to 15% or more) and it is common to ask for all of the NULL (or non-NULL) rows, it is recommended that the ANALYZE data for that index is discarded.

See Also

CREATE INDEX, SELECT

ATTACH DATABASE

Attach a database file

Syntax

Common Usage

```
ATTACH DATABASE 'filename' AS database_name;
```

Description

The `ATTACH DATABASE` command associates the database file *`filename`* with the current database connection under the logical database name *`database_name`*. If the database file *`filename`* does not exist, it will be created. Once attached, all references to a specific database are done via the logical database name, not the filename. All database names must be unique, but (when shared cache mode is not enabled) attaching the same filename multiple times under different database names is properly supported.

The database name `main` is reserved for the primary database (the one that was used to create the database connection). The database name `temp` is reserved for the database that holds temporary tables and other temporary data objects. Both of these database names exist for every database connection.

If the filename `:memory:` is given, a new in-memory database will be created and attached. Multiple in-memory databases can be attached, but they will each be unique. If an empty filename is given (`''`), a temporary file-backed database will be created. Like an in-memory database, each database is unique and all temporary databases are automatically deleted when they are closed. Unlike an in-memory database, file-based temporary databases can grow to large sizes without consuming excessive memory.

All databases attached to a database connection must share the same text encoding as the `main` database. If you attempt to attach a database that has a different text encoding, an SQLite logic error will be returned.

If the `main` database was opened with `sqlite3_open_v2()`, each attached database will be opened with the same flags. If the `main` database was opened read-only, all attached databases will also be read-only.

Associating more than one database to the same database connection enables the execution of SQL statements that reference tables from different database files. Transactions that involve multiple databases are atomic, assuming the main database is not an in-memory database. In that case, transactions within a given database file continue to be atomic, but operations that bridge database files may not be atomic.

If any write operations are performed on any database, a master journal file will be created in association with the `main` database. If the `main` database is located in a read-only area, the master journal file cannot be created and the operation will fail. If some databases are read-only and some are read/write, make sure the `main` database is one of the databases that is located in a read/write area.

Appendix C. SQLite SQL Command Reference. Using SQLite, ISBN: 9781449394592

Any place SQLite expects a table name, it will accept the format *database_name.table_name*. This can be used to refer to a table within a specific database that might otherwise be ambiguous.

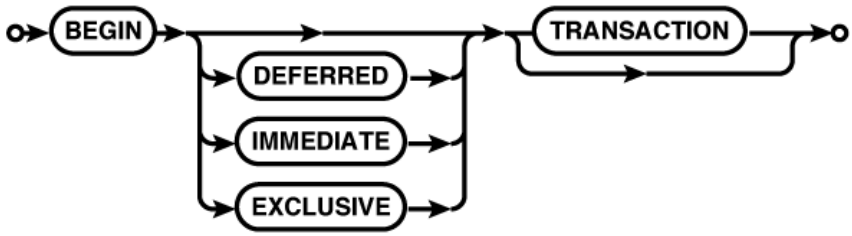
See Also

`DETACH DATABASE`, `encoding` [PRAGMA, Ap F], `temp_store` [PRAGMA, Ap F], `sqlite3_open()` [C API, Ap G]

BEGIN TRANSACTION

Open an explicit transaction

Syntax



Common Usage

```
BEGIN;
BEGIN EXCLUSIVE TRANSACTION;
```

Description

The `BEGIN TRANSACTION` command starts an explicit transaction. Once started, an explicit transaction will remain open until the transaction is either committed using `COMMIT TRANSACTION`, or rolled back using `ROLLBACK TRANSACTION`. Transactions are used to group multiple discrete commands (such as a sequence of `INSERT` commands that insert rows into cross-referenced tables) into a single logical command.

Transactions are ACID-compliant, in that they are atomic, consistent, isolated, and durable. They're an important part of correct database design and database use. For more information, see "Transaction Control Language" on page 51.

All changes and modifications to a database are done within a transaction. Normally, SQLite is in autocommit mode. In this mode, each and every statement that might modify the database is automatically wrapped in its own transaction. Each command begins a transaction, executes the given command statement, and attempts to commit the changes. If any error occurs, the wrapper transaction is rolled back.

The `BEGIN` command turns off autocommit mode, opening a transaction and leaving it open until it is explicitly committed or rolled back. This allows multiple commands (and multiple modifications) to be packaged into a single transaction. Once a `COMMIT` or `ROLLBACK` is issued, the database connection is put back into autocommit mode.

Appendix C. SQLite SQL Command Reference. Using SQLite, ISBN: 9781449394592

Transactions cannot be nested. For that functionality, use `SAVEPOINT`. Executing a `BEGIN` command while the database connection is already in a transaction will result in an error, but will not change the state of the preexisting transaction.

There is a significant cost associated with committing a transaction. In autocommit mode, this cost is seen by every command. In some situations, it can be prudent to wrap several commands into a single transaction. This helps amortize the transaction cost across several statements. When doing large operations, such as bulk inserts, it is not unusual to wrap several hundred, or even a thousand or more `INSERT` commands into a single transaction. The only caution in doing this is that a single error can cause the whole transaction to rollback, so you need to be prepared to re-create all of the rolled back `INSERT` statements.

In SQLite, transactions are controlled by locks. You can specify the locking behavior you want with the modifier `DEFERRED`, `IMMEDIATE`, or `EXCLUSIVE`. The default mode is `DEFERRED`, in which no locks are acquired until they are needed. This allows the highest level of concurrency, but also means the transaction may find itself needing a lock it cannot acquire, and may require a rollback. The `IMMEDIATE` mode attempts to immediately acquire the reserved lock, allowing other connections to continue to read from the database, but reserving the right to elevate itself to write status at any time. Starting an `EXCLUSIVE` transaction will attempt to grab the exclusive lock, allowing full access to the database, but denying access by any other database connection. Higher locking levels means greater certainty that a transaction can be successfully committed at the cost of lower levels of concurrency.

For more information on SQLite's locking and concurrency model, see *http://sqlite.org/lockingv3.html*.

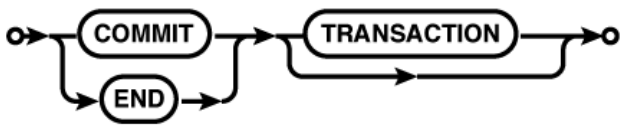
See Also

`COMMIT TRANSACTION`, `ROLLBACK TRANSACTION`, `SAVEPOINT`, `RELEASE SAVEPOINT`

COMMIT TRANSACTION

Finish and commit a transaction

Syntax



Common Usage

```
COMMIT;
```

Description

The `COMMIT TRANSACTION` command attempts to close and commit any changes made during the current transaction. The alias `END TRANSACTION` may also be used. If a `COMMIT` command is made while SQLite is in autocommit mode, an error will be issued.

If the COMMIT is successful, the database will be synchronized and any modifications made within the transaction will become a permanent part of the database record and the database connection will be put back in autocommit mode.

If the commit is not successful, the transaction may or may not be rolled back, depending on the type of error. If the transaction is not rolled back, you can usually just reissue the COMMIT command. If the transaction is rolled back, all modifications made as part of the transaction are lost. You can determine the specific state of the database connection using the sqlite3_get_autocommit() API call, or by trying to issue the BEGIN command. If the database returns a logical error as a result of the BEGIN command, the database is still in a valid transaction. You can also issue the ROLLBACK command, which will either roll back the transaction if it is still in place, or return an error if the transaction was already rolled back.

There is a significant cost associated with committing a transaction. See BEGIN TRANSACTION for more details on how to reduce this cost.

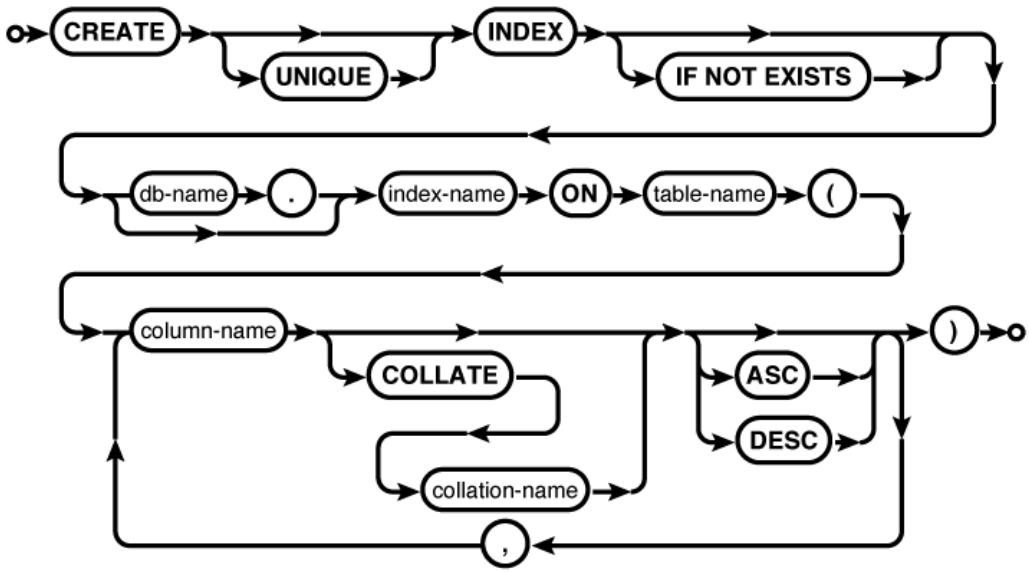
See Also

BEGIN TRANSACTION, ROLLBACK TRANSACTION, END TRANSACTION, sqlite3_get_autocommit() [C API, Ap G]

CREATE INDEX

Define and create a new table index

Syntax



Common Usage

```
CREATE INDEX index_name ON table_name ( column_name COLLATE NOCASE );
CREATE UNIQUE INDEX database_name.index_name ON table_name ( col1, col2 ,... );
```

Description

The `CREATE INDEX` command creates a user-defined index. Upon creation, the index is populated from the existing table data. Once created, the index will be automatically maintained, so that modifications to the referenced table will be reflected in the index. The query optimizer will automatically consider any indexes that have been created. Indexes cannot be created on virtual tables or views.

An index can reference multiple columns, but all of the columns must be from the same table. In the case of multicolumn indexes, the index will be built in the same order as the column listing. For performance-related indexes, the column ordering can be very important. See "Order Matters" on page 109 for more details. The table must be in the same database as the index. To create an index on a temporary table, create the index in the `temp` database.

If a table is dropped, all associated indexes are also dropped. A user-defined index may also be explicitly dropped with the `DROP INDEX` command.

If the optional `UNIQUE` clause is included, the index will not allow inclusion of equivalent index entries. An index entry includes the whole set of indexed columns, taken as a group, so you may still find duplicate column values in a unique multicolumn index. As usual, NULL entries are considered unique from each other, so multiple NULL entries may exist even in a unique single-column index.

An optional collation can be provided for each column. By default, the column's native collation will be used. If an alternate collation is provided, the index can only be used in queries that also specify that collation.

Additionally, each indexed column can specify an ascending (`ASC`) or descending (`DESC`) sort order. By default, all indexed columns will be sorted in an ascending order. Use of descending indexes requires a modern file format. If the database is still using the legacy file format, descending indexes will not be supported and the `DESC` keyword will be silently ignored.

SQLite indexes include a full copy of the indexed data. Be cautious of your database size when indexing columns that consist of large text or BLOB values. Generally, indexes should only be created on columns that have a relatively unique set of values. If any single value appears in more than 10% to 15% of the rows, an index is usually inadvisable. It is almost always unwise to index a Boolean column, or any similar column that holds relatively few values. There is a cost associated with maintaining indexes, so they should only be created when they serve some purpose.

Creating an index that already exists will normally generate an error. If the optional `IF NOT EXISTS` clause is provided, this error is silently ignored. This leaves the original definition (and data) in place.

See Also

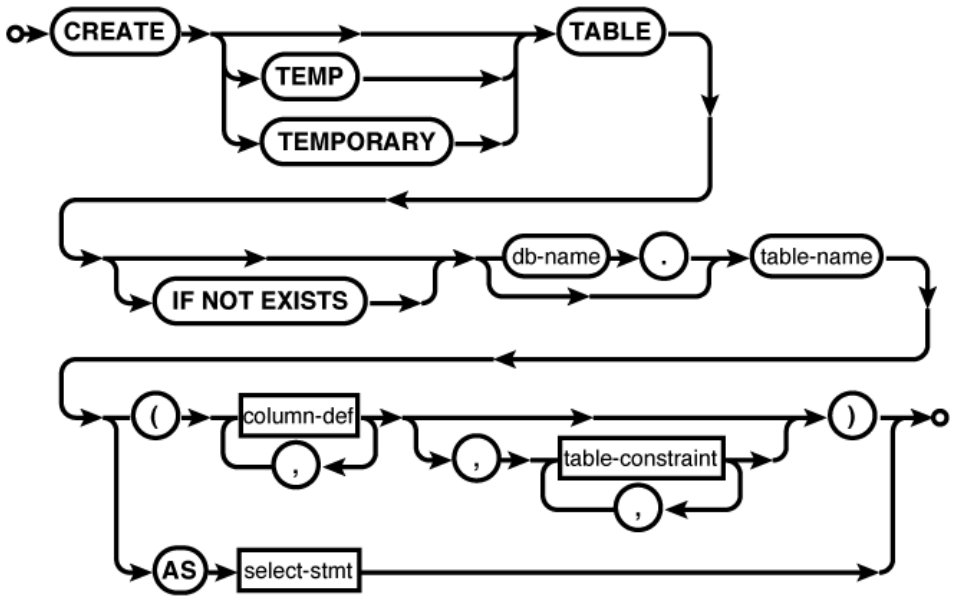
`DROP INDEX`, `ANALYZE`, `REINDEX`, `CREATE TABLE`, `COLLATE` [SQL Expr, Ap D]

Appendix C. SQLite SQL Command Reference. Using SQLite, ISBN: 9781449394592

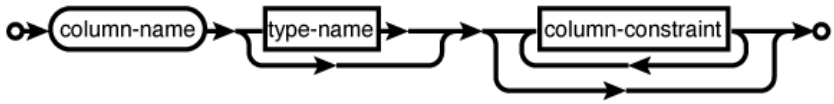
CREATE TABLE

Define and create a new table

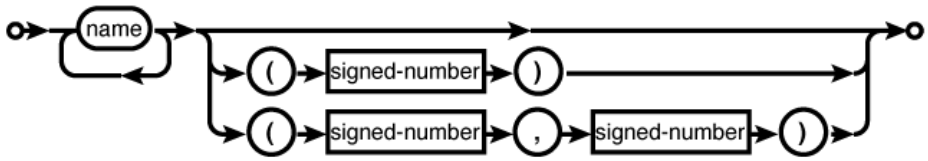
Syntax



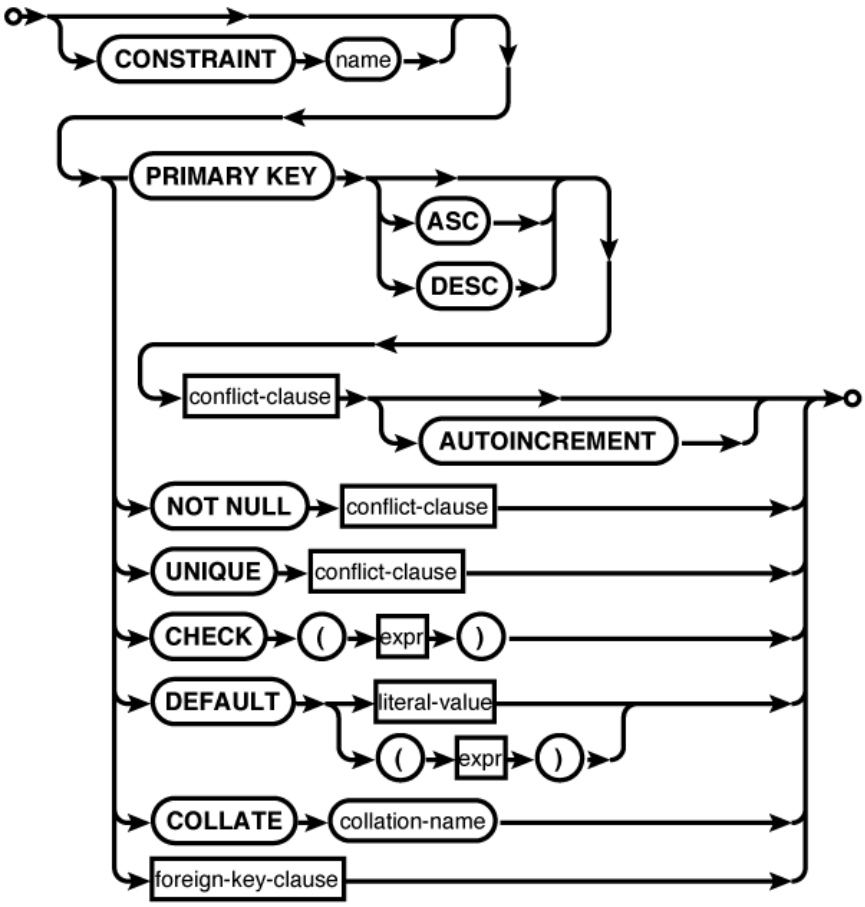
column-def:



type-name:

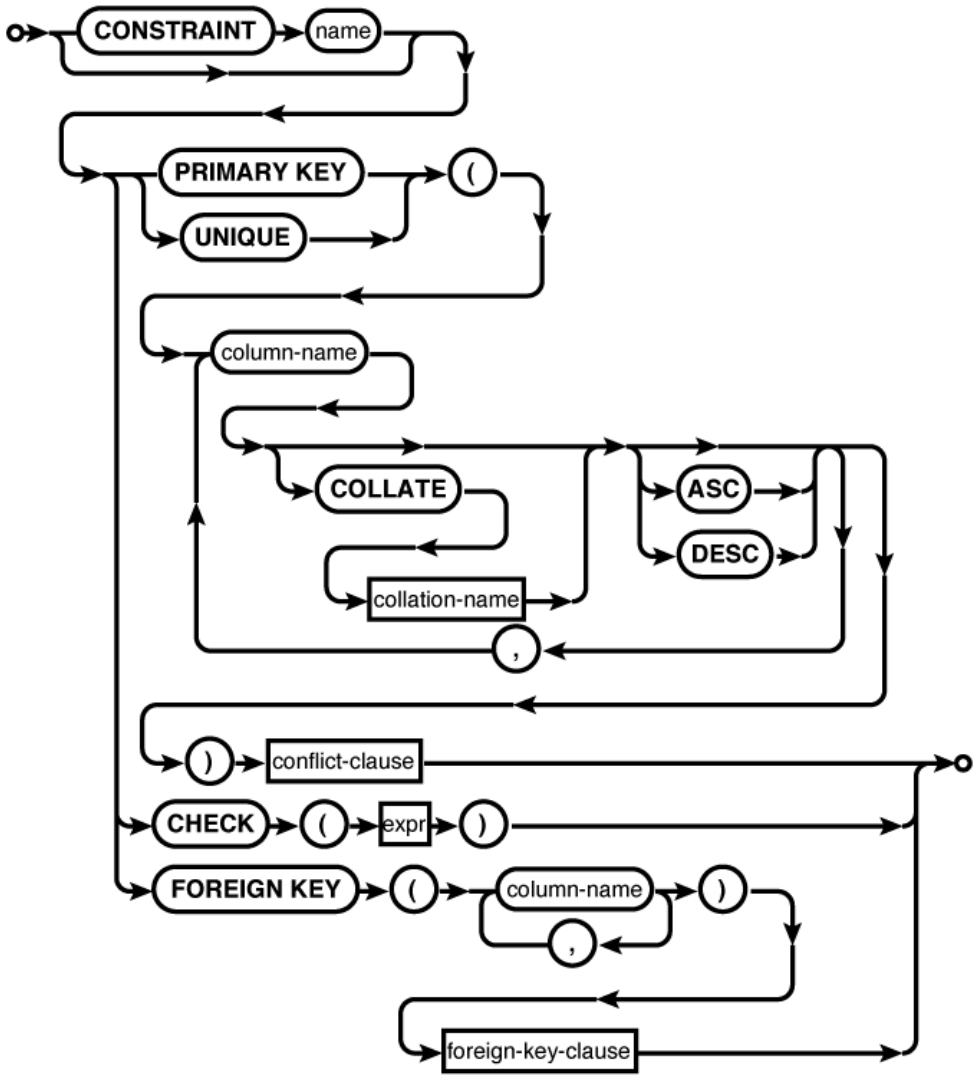


column-constraint:

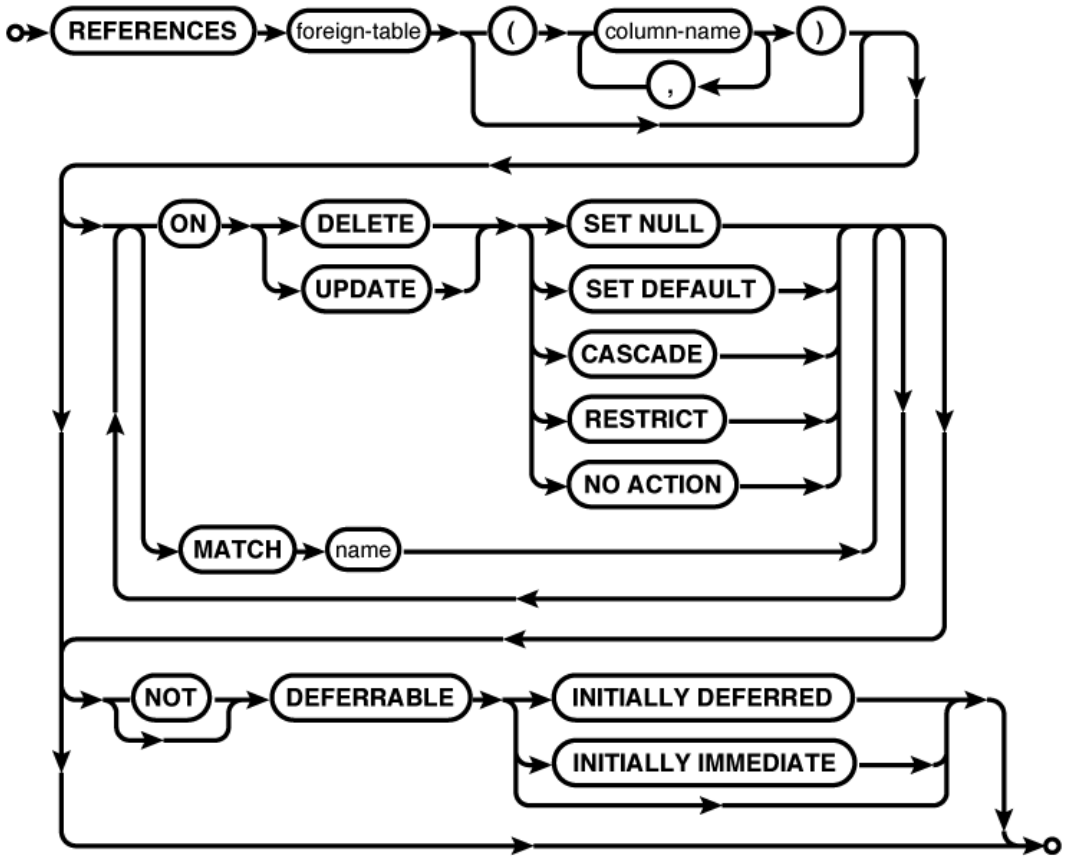


Appendix C. SQLite SQL Command Reference. Using SQLite, ISBN: 9781449394592

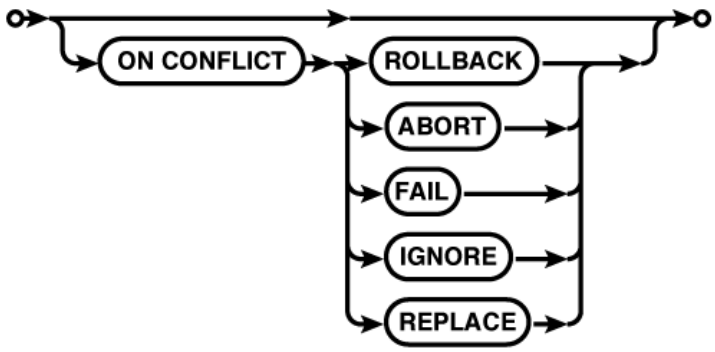
table-constraint:



foreign-key-clause:



conflict-clause:



Common Usage

```
CREATE TABLE database_name.table_name ( c1_name c1_type, c2_name c2_type... );
CREATE TABLE database_name.table_name AS SELECT * FROM... ;
CREATE TABLE tbl ( a, b, c );
CREATE TABLE people ( people_id INTEGER PRIMARY KEY, name TEXT );
CREATE TABLE employee (
   employee_id  INTEGER   PRIMARY KEY   NOT NULL,
   name         TEXT      NOT NULL,
   start_date   TEXT      NOT NULL   DEFAULT CURRENT_DATE,
   parking_spot INTEGER   UNIQUE  );
```

Description

The `CREATE TABLE` command is used to define a new table. It is one of the most complex SQL commands understood by SQLite, though nearly all of the syntax is optional.

A new table can be created in a specific database by qualifying the table name with an explicit database name. If one of the optional keywords `TEMP` or `TEMPORARY` is present, any database name given as part of the table name will be ignored, and the new table will be created in the `temp` database.

Creating a table that already exists will normally generate an error. If the optional `IF NOT EXISTS` clause is provided, this error is silently ignored. This leaves the original definition (and data) in place.

There are two variations of `CREATE TABLE`. The difference is in how the columns are defined. The least common variation uses a simple `AS SELECT` subquery to define the structure and initial contents of the table. The number of columns and the column names will be taken from the result set of the subquery. The rows of the result set will be loaded into the table as part of the table creation process. Because this variation provides no way to define column affinities (typical datatypes), keys, or constraints, it is typically limited to defining "quick and dirty" temporary tables. To quickly create and load structured data, it is often better to create a table using the standard notation and then use an `INSERT...SELECT` command to load the table. The standard notation explicitly defines a list of columns and table constraints.

Basic format

The most common way to define a table structure is to provide a list of column definitions. Column definitions consist of a name and a type, plus zero or more column-level constraint definitions.

The list of column definitions is followed by a list of table-level constraints. For the most part, column-level constraints and table-level constraints are very similar. The main difference is that column constraints apply to the values found in a single column, while table constraints can deal with one or more columns. It is possible to define most column constraints as table-level constraints that only reference a single column. For example, a multicolumn primary key must be defined as a table constraint, but a single-column primary key can be defined as either a table constraint or a column constraint.

The column name is a standard identifier. If nonstandard characters (such as a space or a hyphen) are used, the identifier must be quoted in the `CREATE TABLE` statement as well as any other reference.

The column name is followed by a type indicator. In SQLite, the type is optional, since nearly any column can hold any datatype. SQLite columns do not technically have types, but rather have type affinities. An affinity describes the most favored type for the column and allows SQLite to do implicit conversions in some cases. An affinity does not limit a column to a specific type, however. The use of affinities also accounts for the fact that the type format is extremely flexible, allowing type names from nearly any dialect of SQL. For more specifics on how type affinities are determined and used, see "Column types" on page 36.

If you want to make sure a specific affinity is used, the most straightforward type names are `INT`, `REAL`, `TEXT`, or `BLOB`. SQLite does not use precision or size limits internally. All integer values are signed 64-bit values, all floating-point values are 64-bit values, and all text and BLOB values are variable length.

All tables have an implied root column, known as `ROWID`, that is used internally by the database to index and store the database table structure. This column is not normally displayed or returned in queries, but can be accessed directly using the name `ROWID`, `_ROWID_`, or `OID`. The alternate names are provided for compatibility with other database engines. Generally, `ROWID` values should never be used or manipulated directly, nor should the `ROWID` column be directly used as a table key. To use a `ROWID` as a key value, it should be aliased to a user-defined column. See "PRIMARY KEY constraint" on page 314.

Column constraints

Each column definition can include zero or more column constraints. Column constraints follow the column type indicator; there is no comma or other delimiter between basic column definitions and the column constraints. The constraints can be given in any order.

Most of the column constraints are easy to understand. The `PRIMARY KEY` constraint is a bit unique, however, and is discussed below, in its own section.

The `NOT NULL` constraint prohibits the column from containing NULL entries. The `UNIQUE` constraint requires all the values of the column to be unique. An automatic unique index will be created on the column to enforce this constraint. Be aware that `UNIQUE` does not imply `NOT NULL`, and unique columns are allowed to have more than one NULL entry. This means there is a tendency for columns with a `UNIQUE` constraint to also have a `NOT NULL` constraint.

The `CHECK` constraint provides an arbitrary user-defined expression that must remain true. The expression can safely access any column in the row. The `CHECK` constraint is very useful to enforce specific data formats, ranges or values, or even specific datatypes. For example, if you want to be absolutely sure nothing but integer values are entered into a column, you can add a constraint such as:

```
CHECK ( typeof( column_name ) == 'integer' )
```

The `DEFAULT` constraint defines a default value for the column. This value is used when an `INSERT` statement does not include a specific value for this column. A `DEFAULT` can either be a literal value or, if enclosed in parentheses, an expression. Any expression must evaluate to a constant value. You can also use the special values `CURRENT_TIME`, `CURRENT_DATE`, or `CURRENT_TIMESTAMP`. These will insert an appropriate text value indicating the time the row was first created. If no `DEFAULT` constraint is given, the default value will be NULL.

The COLLATION constraint is used to assign a specific collation to a column. This not only defines the sort order for the column, it also defines how values are tested for equality (which is important for things such as UNIQUE constraints). SQLite includes three built-in collations: BINARY (the default), NOCASE, and RTRIM. BINARY treats all values as binary data that must match exactly. NOCASE is similar to binary, only it is case-insensitive for ASCII text values (in specific, character codes < 128). Also included is RTRIM (right-trim), which is like BINARY, but will trim any trailing whitespace from TEXT values before doing comparisons.

Finally, columns can contain a REFERENCES foreign key constraint. If given as a column constraint, the foreign table reference can contain no more than one foreign column name. If no column references are given, the foreign table must have a single-column primary key. For more information on foreign keys, see the section "Foreign Keys" on page 89. Note that a column-level foreign key constraint does not actually contain the words FOREIGN KEY. That syntax is for table-level foreign key constraints.

Table constraints

Generally, the table-level constraints are the same as the column-level constraints, except that they operate across more than one column. In most cases, table-level constraints have similar syntax to their column-level counterparts, with the addition of a list of columns that are applied to the constraint.

The UNIQUE table constraint requires that each group of column values must be UNIQUE from all the other groups within the table. In the case of a multicolumn UNIQUE constraint, any individual column is allowed to have duplicate values, it is only the group of column values, taken as a whole, that must remain unique. Both UNIQUE and PRIMARY KEY multicolumn constraints can define individual column collations and orderings that are different from the individual column collations.

The table-level CHECK constraint is identical to the column-level CHECK constraint. Both forms are allowed to use an arbitrary expression that references any column in the row.

Finally, multicolumn foreign keys are defined with the FOREIGN KEY constraint. The list of local table columns must be the same size, and in the same order, as the foreign column list provided by the REFERENCES clause. For more information on foreign keys, see "Foreign Keys" on page 89.

PRIMARY KEY constraint

The PRIMARY KEY constraint is used to define the primary key (or PK) for the table. From a database design and theory standpoint, it is desirable for every table to have a primary key. The primary key defines the core purpose of the table by defining the specific data points that make each row a unique and complete record.

From a practical standpoint, SQL does not require a table to have a PK. In fact, SQL does not require that rows within a table be unique. Nonetheless, there are some advantages to defining a primary key, especially when using foreign keys. In most cases a foreign key in one table will refer to the primary key of another table, and explicitly defining a primary key can make it easier to establish this relationship. SQLite also provides some additional features for single-column primary keys.

There can be only one PRIMARY KEY constraint per table. It can be defined at either the column level or the table level, but each table can have only one. A PRIMARY KEY constraint implies a

UNIQUE constraint. As with a standalone UNIQUE constraint, this will cause the creation of an automatic unique index (with one exception). In most database systems, PRIMARY KEY also implies NOT NULL, but due to a long-standing bug, SQLite allows the use of NULL entries in a primary key column. For proper behavior, be sure to define at least one column of the primary key to be NOT NULL.

If a column has the type identifier INTEGER (it can be upper- or lowercase, but must be the exact word "integer"), an ascending collation (the default), and has a single-column PRIMARY KEY constraint, then that column will become an alias for the ROWID column. Behind the scenes, this makes an INTEGER PRIMARY KEY column the root column, used internally to index and store the database table. Using a ROWID alias allows for very fast row access without requiring a secondary index. Additionally, SQLite will automatically assign an unused ROWID value to any row that is inserted without an explicit column value.

Columns defined as INTEGER PRIMARY KEY can really truly hold only integer values. Additionally, unlike other primary key columns, they have an inherent NOT NULL constraint. Default values are assigned using the standard ROWID allocation algorithm. This algorithm will automatically assign a value that is one larger than the largest currently used ROWID value. If the maximum value is met, a random (unused) ROWID value will be chosen. As rows are added and removed from a table, this allows ROWID values to be recycled.

While recycling values is not a problem for internal ROWID values, it can cause problems for reference values that might be lurking elsewhere in the database. To avoid problems, the keyword AUTOINCREMENT can be used with an INTEGER PRIMARY KEY to indicate that automatically generated values should not be recycled. Default values assigned by AUTOINCREMENT will be one larger than the largest ROWID value that was ever used, but don't depend on each and every value being used. If the maximum value is reached, an error is returned.

When using a ROWID alias to automatically generate keys, it is a common practice to insert a new row and call the SQL function last_insert_rowid(), or the C function sqlite3_last_insert_rowid(), to retrieve the ROWID value that was just assigned. This value can be used to insert or update rows that reference the newly inserted row. It is also always possible to insert a row with a specific ROWID (or ROWID alias) value.

Conflict clause

Nearly every column constraint and table constraint can have an optional conflict resolution clause. This clause can be used to specify what action SQLite takes if a command attempts to violate that particular constraint. Constraint violations most commonly happen when attempting to insert or update invalid row values.

The default action is ON CONFLICT ABORT, which will attempt to back-out any changes made by the command that caused the constraint violation, but will otherwise attempt to leave any current transaction in place and valid. For more information on the other conflict resolution choices, see UPDATE. Note that the conflict resolution clause in UPDATE and INSERT applies to the actions taken by the UPDATE and INSERT commands themselves. Any conflict resolution clause found in a CREATE TABLE statement is applied to any command operating on the table.

See Also

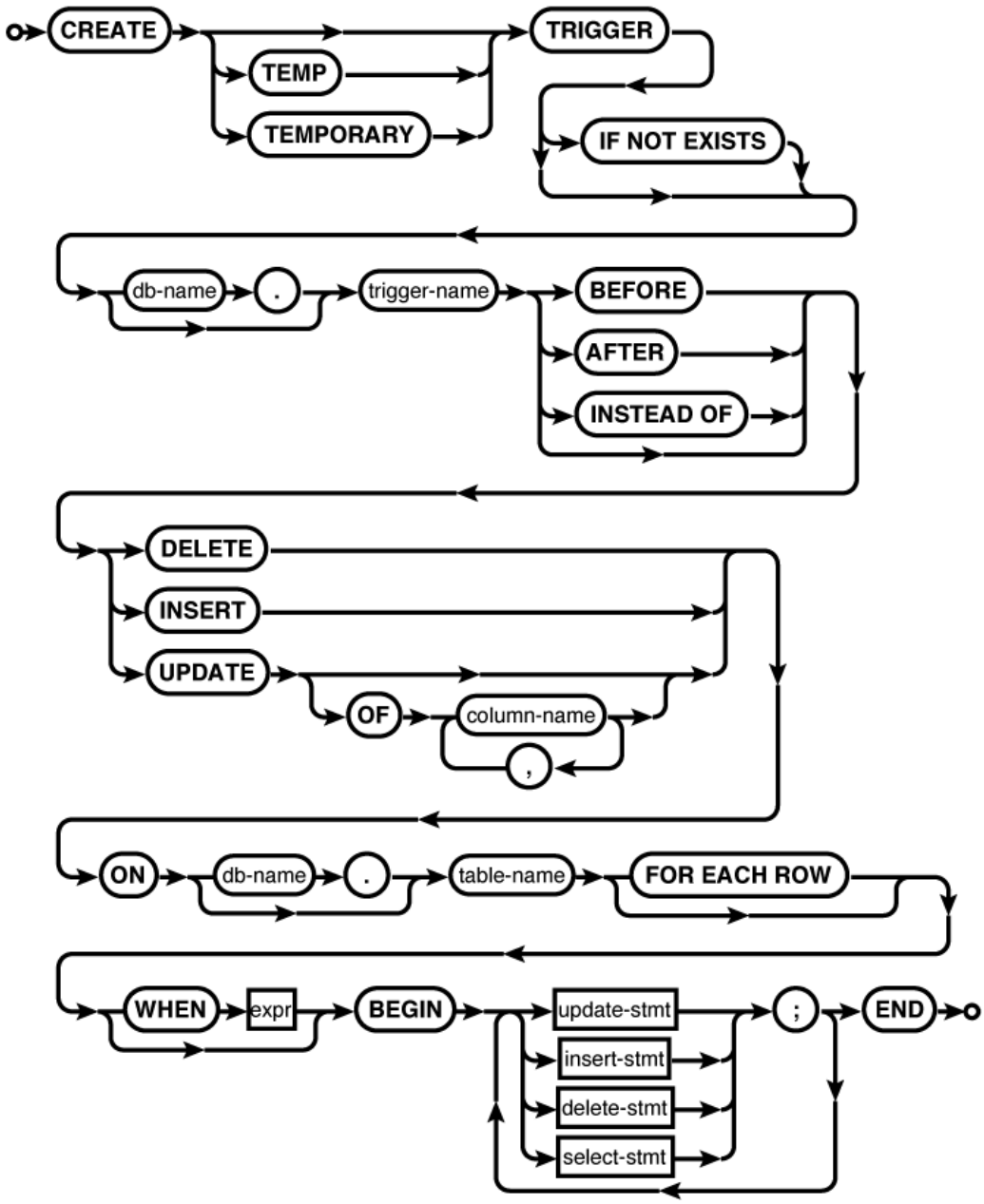
DROP TABLE, INSERT, UPDATE, CREATE INDEX

Appendix C. SQLite SQL Command Reference. Using SQLite, ISBN: 9781449394592

CREATE TRIGGER

Create a new trigger action

Syntax



Common Usage

```
CREATE TRIGGER database_name.trigger_name BEFORE INSERT ON table_name FOR EACH ROW
  BEGIN stmt1; stmt2; END;
CREATE TRIGGER access_audit BEFORE UPDATE ON access FOR EACH ROW
  BEGIN
    INSERT INTO audit_trail VALUES ( OLD.level, NEW.level, CURRENT_TIMESTAMP );
  END;
```

Description

The `CREATE TRIGGER` command creates a trigger and binds it to a table or view. When the conditions defined by the trigger are met, the trigger will "fire," automatically executing statements found in the trigger body (the part between `BEGIN` and `END`). A table or view may have any number of triggers associated with it, including multiple triggers of the same type.

If the optional `TEMP` or `TEMPORARY` keyword is present, a trigger will be created in the `temp` database. A trigger can also be made temporary by qualifying the trigger name with the database name `temp`. If the trigger name is qualified with a database name, specifying `TEMP` or `TEMPORARY` will result in an error, even if the given database name is `temp`.

Temporary triggers can be attached to either temporary or standard (nontemporary) tables. A specific table instance can be chosen by qualifying the table name with a database name. In all other cases, the trigger and the table should be in the same database. Either the trigger name or the table name can be qualified with a database name (or both, if they match).

Triggers associated with tables may be `BEFORE` or `AFTER` triggers. If no time is specified, `BEFORE` is used. The timing indicates if the trigger fires before or after the defined trigger action. In both cases, the action is verified before the trigger is fired. For example, a `BEFORE INSERT` trigger will not fire if the insert will cause a constraint violation.

The trigger action can be either a `DELETE`, `INSERT`, or `UPDATE` statement that gets run against the trigger's table. In the case of `UPDATE`, the trigger can fire when any column is updated, or only when one or more columns from the specified list is updated.

Triggers associated with views must be `INSTEAD OF` triggers. The default timing for views is still `BEFORE`, so the `INSTEAD OF` must be specified. As the name indicates, `INSTEAD OF` triggers fire in the place of the defined action. Although views are read-only in SQLite, defining one or more `INSTEAD OF DELETE`, `INSERT`, or `UPDATE` trigger will allow those commands to be run against the view. Very often, views will have a whole series of `INSTEAD OF` triggers to deal with different combinations of column updates.

The SQL standard defines both `FOR EACH ROW` as well as `FOR EACH STATEMENT` triggers. SQLite only supports `FOR EACH ROW` triggers, which fire once for each row affected by the specified condition. This makes the `FOR EACH ROW` clause optional in SQLite. Some popular databases that support both types of triggers will default to `FOR EACH STATEMENT` triggers, however, so explicit use of the `FOR EACH ROW` clause is recommended.

Triggers also have an optional `WHEN` clause that is used to control whether the trigger actually fires or not. Don't underestimate the `WHEN` clause. In many cases, the logic in the `WHEN` clause is more complex than the trigger body.

The trigger body itself consists of one or more INSERT, UPDATE, DELETE, or SELECT statements. The first three commands can be used in the normal way. A SELECT statement can be used to call user-defined functions. Any results returned by a standalone SELECT statement will be ignored. Table identifiers within the trigger body cannot be qualified with a database name. All table identifiers must be from the same database as the trigger table.

Both the WHEN clause and the trigger body have access to some additional column qualifiers. Columns associated with the trigger table (or view) may be qualified with the pseudo-identifier NEW (in the case of INSERT and UPDATE triggers) or OLD (in the case of UPDATE and DELETE triggers). These represent the before and after values of the row in question and are only valid for the current row that caused the trigger to fire.

Commands found in a trigger body can also use the RAISE expression to raise an exception. This can be used to ignore, roll back, abort, or fail the current row in an error situation. For more information, see RAISE and UPDATE.

There are some additional limits on trigger bodies. Within a trigger body, UPDATE and DELETE commands cannot use index overrides (INDEXED BY, NOT INDEXED), nor is the ORDER BY...LIMIT syntax supported (even if support has been properly enabled). The INSERT...DEFAULT VALUES syntax is also unsupported. If a trigger is fired as the result of a command with an explicit ON CONFLICT clause, the higher-level conflict resolution will override any ON CONFLICT clause found in a trigger body.

If a trigger modifies rows from the same table it is attached to, the use of AFTER triggers is strongly recommended. If a BEFORE trigger modifies the rows that are part of the original statement (the one that caused the trigger to fire) the results can be undefined. Also, the NEW.ROWID value is not available to BEFORE INSERT triggers unless an explicit value has been provided.

If a table is dropped, all of its triggers are automatically dropped. Similarly, if a table is renamed (via ALTER TABLE), any associated triggers will be updated. However, dropping or altering a table will not cause references found in a trigger body to be updated. If a table is dropped or renamed, make sure any triggers that reference it are updated as well. Failing to do so will cause an error when the trigger is fired.

Creating a trigger that already exists will normally generate an error. If the optional IF NOT EXISTS clause is provided, this error is silently ignored. This leaves the original definition (and data) in place.

One final note. Some of the syntax and many of the functional limitations of CREATE TRIGGER are checked at execution, not at creation. Just because the CREATE TRIGGER command returned without error doesn't mean the trigger description is valid. It is strongly suggested that all triggers are verified and tested. If a trigger encounters an error, that error will be bubbled up to the statement that caused the trigger to fire. This can cause perplexing results, such as commands producing errors about tables or columns that are not part of the original statement. If a command is producing an unexplained or odd error, check to make sure there are no faulty triggers associated with any of the tables referenced by the command.

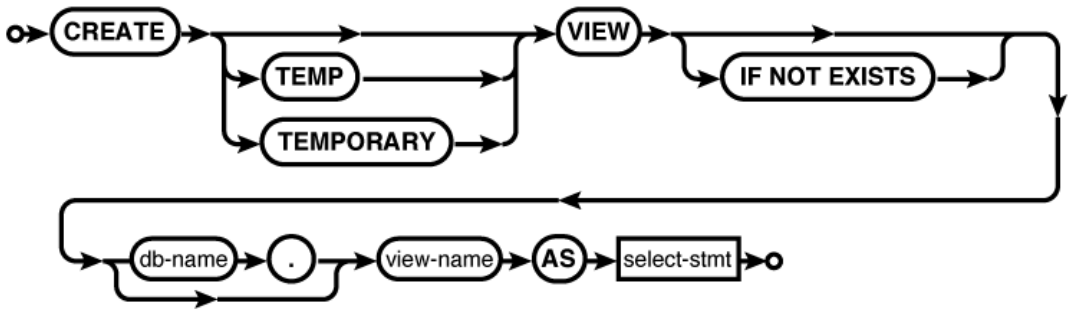
See Also

DROP TRIGGER, CREATE TABLE, CREATE VIEW, INSERT, UPDATE, DELETE, RAISE [SQL Expr, Ap D]

CREATE VIEW

Create a new view

Syntax



Common Usage

```
CREATE VIEW database_name.view_name AS SELECT...;
```

Description

The `CREATE VIEW` statement establishes a new view within the named database. A view acts as a prepackaged subquery statement, and can be accessed and referenced as if it were a table. A view does not actually instance the data, but is dynamically generated each time it is accessed.

If the optional `TEMP` or `TEMPORARY` keyword is present, the view will be created in the `temp` database. Specifying either `TEMP` or `TEMPORARY` in addition to an explicit database name will result in an error, unless the database name is `temp`.

Temporary views may access tables from other attached databases. All nontemporary views are limited to referencing data sources from within their own database.

Creating a view that already exists will normally generate an error. If the optional `IF NOT EXISTS` clause is provided, this error is silently ignored. This leaves the original definition in place.

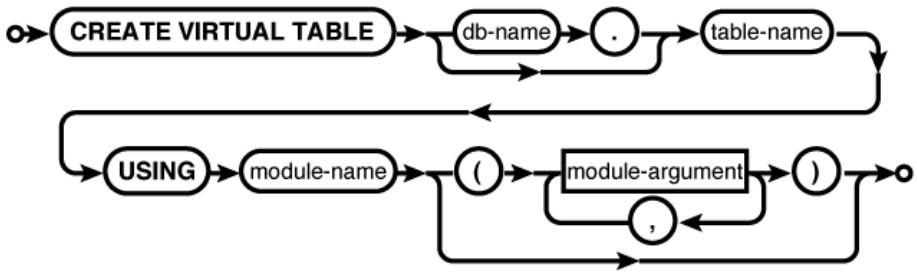
See Also

`DROP VIEW`, `CREATE TABLE`, `SELECT`

CREATE VIRTUAL TABLE

Create a new virtual table

Syntax



Common Usage

```
CREATE VIRTUAL TABLE database_name.table_name USING weblog( access.log );
CREATE VIRTUAL TABLE database_name.table_name USING fts3( );
```

Description

The `CREATE VIRTUAL TABLE` command creates a virtual table. Virtual tables are data sources that are defined by code and can represent highly optimized data sources or external data sources. The standard SQLite distribution includes virtual table implementations for Full Text Search, as well as an R*Tree-based indexing system.

Virtual tables are covered in detail in Chapter 10.

A virtual table is removed with the standard `DROP TABLE` command.

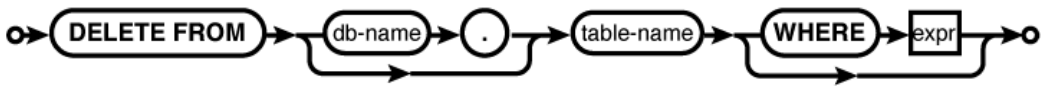
See Also

`sqlite3_create_module()` [C API, Ap G], `DROP TABLE`

DELETE

Delete rows from a table

Syntax

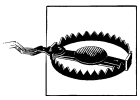


Common Usage

```
DELETE FROM database_name.table_name;
DELETE FROM database_name.table_name WHERE id = 42;
```

Description

The `DELETE` command permanently removes rows from a table. Any row that satisfies the `WHERE` expression will be removed. A `WHERE` condition that causes no rows to be deleted is not considered an error. If no `WHERE` condition is provided, it is assumed to always be true, and every row in the table will be deleted.



A DELETE command with no WHERE clause will delete *every* row in a table.

If no WHERE clause is provided, there are some situations when SQLite can simply truncate the whole table. This is much faster than deleting every row individually, but it skips any per-row processing. Truncation will only happen if the table has no triggers and is not part of a foreign key relationship (assuming foreign key support is enabled). Truncation can also be disabled by having an authorizer return SQLITE_IGNORE for the delete operation (see sqlite3_set_authorizer()).

If the SQLite library has been compiled with the optional SQLITE_ENABLE_UPDATE_DELETE_LIMIT directive, an optional ORDER BY...LIMIT clause may be used to delete a specific number of rows. See the SQLite website for more details.

When a DELETE appears within a trigger body, additional limitations apply. See CREATE TRIGGER.

Deleting data from a table will not decrease the size of the database file unless auto-vacuum mode is enabled. To recover space previously taken up by deleted data, the VACUUM command must be run.

See Also

INSERT, UPDATE, VACUUM, auto_vacuum [PRAGMA, Ap F], CREATE TRIGGER

DETACH DATABASE

Detach a database file

Syntax

Common Usage

```
DETACH DATABASE database_name;
```

Description

The DETACH DATABASE command detaches and dissociates a named database from a database connection. If the same file has been attached multiple times, this will only detach the named attachment. If the database is an in-memory or temporary database, the database will be destroyed and the contents will be lost.

You cannot detach the main or temp databases. The DETACH command will fail if issued inside a transaction.

See Also

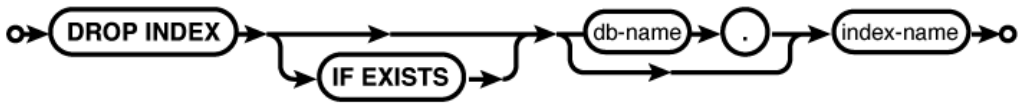
ATTACH DATABASE

Appendix C. SQLite SQL Command Reference. Using SQLite, ISBN: 9781449394592

DROP INDEX

Delete a table index from a database

Syntax



Common Usage

```
DROP INDEX database_name.index_name;
```

Description

The `DROP INDEX` command deletes an explicitly created index. The index and all the data it contains is deleted from the database. The table the index references is not modified. You cannot drop automatically generated indexes, such as those that enforce unique constraints declared in table definitions.

Dropping an index that does not exist normally generates an error. If the optional `IF EXISTS` clause is provided, this error is silently ignored.

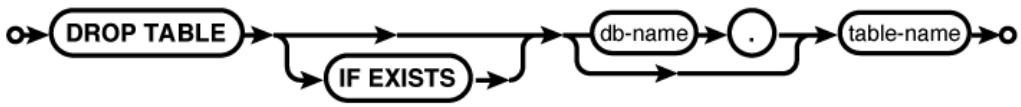
See Also

`CREATE INDEX`, `CREATE TABLE`, `DROP TABLE`

DROP TABLE

Delete a table from a database

Syntax



Common Usage

```
DROP TABLE database_name.table_name;
```

Description

The `DROP TABLE` command removes a table from a database. The table and all the data it contains are permanently removed from the database. Any associated indexes and triggers are also removed. Views that might reference the table are not removed. Delete triggers will not be fired.

The `DROP TABLE` command may also be used to remove virtual tables. In that case, a destroy request is sent to the table module, which is free to do as it sees fit.

If foreign keys are enabled, the DROP TABLE command will perform the equivalent of a DELETE for each row in the table. This happens after any associated triggers have been dropped, so this will not cause any delete triggers to fire. If any immediate key constraints are violated, the DROP TABLE command will fail. If any deferred constraints are violated, an error will be returned when the transaction is committed.

Unless the database is in auto-vacuum mode, dropping a table will not cause the database file to shrink in size. The database pages used to hold the table data will be placed on the free list, but they will not be released. The database must be vacuumed to release the free database pages.

Dropping a table that does not exist normally generates an error. If the optional IF EXISTS clause is provided, this error is silently ignored.

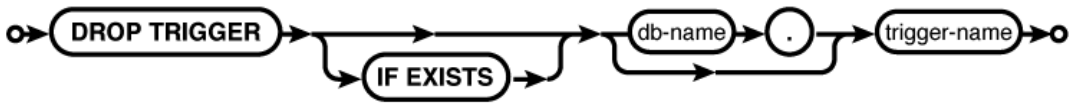
See Also

CREATE TABLE, ALTER TABLE, DROP INDEX, DROP TRIGGER, auto_vacuum [PRAGMA, Ap F], VACUUM

DROP TRIGGER

Delete a trigger action from a database

Syntax



Common Usage

```
DROP TRIGGER database_name.trigger_name;
```

Description

The DROP TRIGGER command removes a trigger from the database. A trigger will also be removed when the associated table is removed.

Dropping a trigger that does not exist normally generates an error. If the optional IF EXISTS clause is provided, this error is silently ignored.

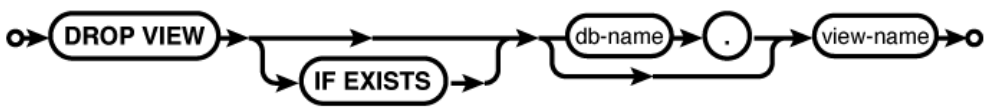
See Also

CREATE TRIGGER, DROP TABLE

DROP VIEW

Delete a view from a database

Syntax



Common Usage

```
DROP VIEW database_name.view_name;
```

Description

The `DROP VIEW` command removes a view from the database. Although the view will no longer be available, none of the referenced data is altered in any way. Dropping a view will also remove any associated triggers.

Dropping a view that does not exist will normally generate an error. If the optional `IF EXISTS` clause is provided, this error is silently ignored.

See Also

`CREATE VIEW`, `CREATE TABLE`, `DROP TRIGGER`

END TRANSACTION

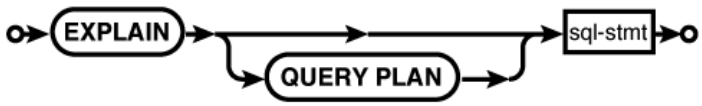
Finish and commit a transaction

See: `COMMIT TRANSACTION`

EXPLAIN

Explain the query plan

Syntax



Common Usage

```
EXPLAIN  INSERT ...;
EXPLAIN QUERY PLAN  SELECT ...;
```

Description

The `EXPLAIN` command offers insight into the internal database operation. Placing `EXPLAIN` in front of any SQL statement (other than itself) returns information about how SQLite would execute the given SQL statement. The SQL statement is not actually executed.

By itself, `EXPLAIN` will return a result set that describes the internal VDBE instruction sequence used to execute the provided SQL statement. A fair amount of knowledge is required to understand the output.

The full `EXPLAIN QUERY PLAN` variant will return a high-level summary of the query plan using English-like explanations of how the query will be assembled and if data will be accessed by a table scan or by index lookup. The `EXPLAIN QUERY PLAN` command is most useful for tuning `SELECT` statements and adjusting index placement.

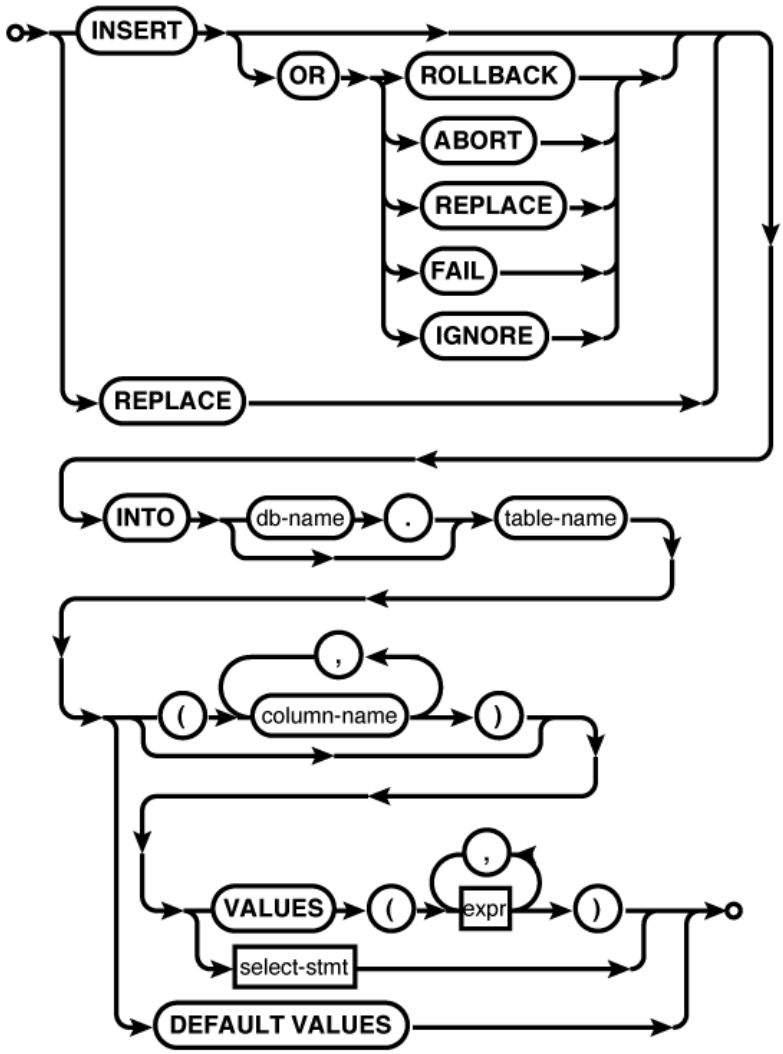
These commands exist to help database administrators and developers understand how SQLite is processing SQL commands. They are designed for interactive tuning and adjustments. It is

not recommended that applications programmatically query or utilize this data, as both the data and the format may change from one version of SQLite to the next.

INSERT

Insert new rows into a table

Syntax



Common Usage

```
INSERT INTO database_name.table_name ( col1, col2 ) VALUES ( val1, val2 );
INSERT INTO table_name VALUES ( val1, val2, val3... );
INSERT INTO table_name ( col1, col2 ) SELECT c1, c2 FROM...;
INSERT INTO table_name DEFAULT VALUES;
INSERT OR IGNORE INTO table_name ( col1, col2 ) VALUES ( val1, val2 );
REPLACE INTO table_name ( col1, col2 ) VALUES ( val1, val2 );
```

Description

The INSERT command adds new rows to tables. An individual INSERT command can only insert rows into one table, and in most cases can only insert one row at a time. There are several variants of the INSERT command. The primary differences relate to how the columns and values are specified.

The basic format of the INSERT command starts with the command word INSERT, followed by an optional conflict resolution clause (OR ROLLBACK, etc.). The INSERT conflict resolution clause is identical to the one found in the UPDATE command. See UPDATE for more information on the different conflict resolution options and how they behave. The command word REPLACE can also be used, which is just a shortcut for INSERT OR REPLACE. This is discussed in more detail below.

After the conflict clause, the command declares which table it is acting upon. This is generally followed by a list of columns in parentheses. This column list defines which columns will have values set in the newly inserted row. If no column list is given, a default column list is assumed to include all of the table's columns, in order, as they appear in the table definition. A default list will not include the raw ROWID column, but any ROWID alias (INTEGER PRIMARY KEY) column is included. If an explicit list of columns is given, the list may contain any column (including the ROWID column) in any order.

Following the column list is a description of the values to insert into the new row. The values are most commonly defined by the keyword VALUES, followed by an explicit list of values in parentheses. The values are matched to columns by position. No matter how the column list is defined, the column list and the value list must have the same number of entries so that one value can be matched to each column in the column list.

The INSERT values can also be defined with a subquery. Using a subquery is the only case when a single INSERT command can insert more than one row. The result set generated by the subquery must have the same number of columns as the column list. As with the value list, the values of the subquery result set are matched to the insert columns by position.

Any table column that does not appear in the insert column list is assigned a default value. Unless the table definition says otherwise, the default value is a NULL. If you have a ROWID alias column that you want assigned an automatic value, you must use an explicit column list, and that list must not include the ROWID alias. If a column is contained in the column list, either explicitly, or by default, a value must be provided for that column. There is no way to specify a default value except by leaving the column out of the column list, or knowing what the default value is and explicitly inserting it.

Alternatively, the column list and value list can be skipped all together. The DEFAULT VALUES variant provides neither a column list nor a source of values and can be used to insert a new row that consists entirely of default values.

Because a typical INSERT command only allows a single row to be inserted, it is not uncommon to have a string of INSERT commands that are used to bulk-load or otherwise populate a new table. Like any other command, each INSERT is normally wrapped in its own transaction. Committing and synchronizing this transaction can be quite expensive, often limiting the number of inserts to a few dozen a second.

Appendix C. SQLite SQL Command Reference. Using SQLite, ISBN: 9781449394592

Due of the expense associated with committing a transaction, it is very common to batch multiple inserts into a single transaction. Especially in the case of bulk inserts, it is not uncommon to batch 1,000 or even 10,000 or more `INSERT` commands into a single transaction, allowing a much higher insert rate. Just understand that if an error is encountered, there are situations where the whole transaction will be rolled back. While this may be acceptable for loading bulk data from a file (that can simply be rerun), it may not be acceptable for data that is inserted from a real-time data stream. Batch transactions greatly speed things up, but they can also make it significantly more difficult to recover from an error.

One final word on the `INSERT OR REPLACE` command. This type of command is frequently used in event-tracking systems where a "last seen" timestamp is required. When an event is processed, the system needs to either insert a new row (if this type of event has never been seen before) or it needs to update an existing record. While the `INSERT OR REPLACE` variant seems perfect for this, it has some specific limitations. Most importantly, the command truly is an "insert or replace" and not an "insert or update." The `REPLACE` option fully deletes any old rows before processing the `INSERT` request, making it ineffective to update a subset of columns. In order to effectively use the `INSERT OR REPLACE` option, the `INSERT` needs to be capable of completely replacing the existing row, and not simply updating a subset of columns.

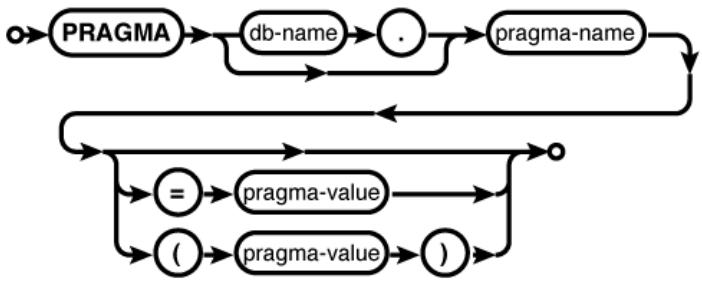
See Also

`UPDATE`, `BEGIN TRANSACTION`

PRAGMA

Look up or modify an SQLite configuration

Syntax



Common Usage

```
PRAGMA page_size;
PRAGMA cache_size = 5000;
PRAGMA table_info( table_name );
```

Description

The `PRAGMA` command tunes and configures SQLite's internal components. It is a bit of a catchall command, used to configure or query configuration parameters for both the database engine and database files. It can also be used to query information about a database, such as a list of tables, indexes, column information, and other aspects of the schema.

Appendix C. SQLite SQL Command Reference. Using SQLite, ISBN: 9781449394592

The PRAGMA command is the only command, outside of SELECT, that may return multiple rows.

Appendix F covers the different PRAGMA commands in detail.

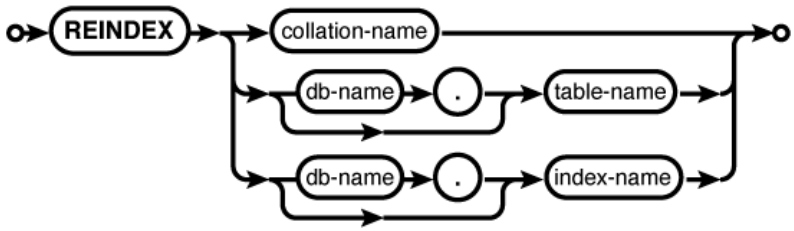
See Also

Appendix F

REINDEX

Rebuild an index from source data

Syntax



Common Usage

```
REINDEX collation_name;
REINDEX database_name.table_name;
REINDEX database_name.index_name;
```

Description

The REINDEX command deletes the data within an index and rebuilds the index structure from the source table data. The table referenced by the index is not changed.

REINDEX is most frequently used when the definition of a collation sequence has changed and all of the indexes that use that collation must be rebuilt. This ensures that the index order correctly matches the order defined by the collation.

If a collation name is provided, all indexes that use that collation, in all attached databases, will be reindexed. If a table name is given, all the indexes associated with that table will be reindexed. If a specific index name is given, just that index will be rebuilt.

See Also

CREATE INDEX, DROP INDEX

RELEASE SAVEPOINT

Remove and release save-point from transaction log

Syntax

Common Usage

```
RELEASE savepoint_name;
```

Description

The `RELEASE SAVEPOINT` command removes a save-point from the transaction log. It indicates that any modifications made since the named save-point was set have been accepted by the application logic.

Normally, a `RELEASE` will not alter the database or transaction log, other than removing the save-point marker. Releasing a save-point will not commit any modifications to disk, nor will it make those changes available to other database connections accessing the same database. Changes bounded by a released save-point may still be lost if the transaction is rolled back to a prior save-point, or if the whole transaction is rolled back.

The one exception to this rule is if the named save-point was set outside of a transaction, causing an implicit transaction to be started. In that case, releasing the save-point will cause the whole transaction to be committed.

See Also

`SAVEPOINT`, `ROLLBACK TRANSACTION`, `COMMIT TRANSACTION`, `BEGIN TRANSACTION`

REPLACE

Delete and reinsert an existing row

Description

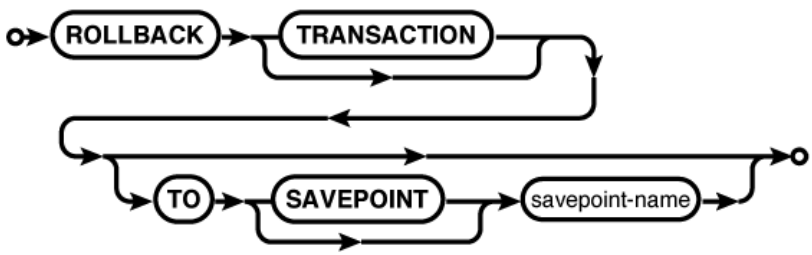
The `REPLACE` command is an alias for the `INSERT OR REPLACE` variant of the `INSERT` command.

See: `INSERT`

ROLLBACK TRANSACTION

Undo part or all of the current transaction

Syntax



Common Usage

```
ROLLBACK;
ROLLBACK TO SAVEPOINT savepoint_name;
```

Description

The ROLLBACK command is used to roll back a transaction state. This is analogous to an undo function for transactions.

There are two forms of ROLLBACK. The most basic form has no TO clause and causes the entire transaction to be rolled back. Any and all changes and modifications made to the database as part of the transaction are reverted, the transaction is released, and the database connection is put back into autocommit mode with no active transaction.

If a TO clause is provided, the transaction is rolled back to the state it was in just *after* the named save-point was created. The named save-point will remain on the save-point stack. You can roll back to any save-point, but if more than one save-point exists with the same name, the most recent save-point will be used. After rolling back to a save-point, the original transaction is still active.

If the named save-point was created outside of a transaction (causing an implicit transaction to be started) the whole transaction will be rolled back, but the save-point and transaction will remain in place.

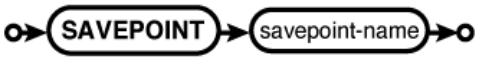
See Also

BEGIN TRANSACTION, COMMIT TRANSACTION, SAVEPOINT, RELEASE SAVEPOINT

SAVEPOINT

Place a save-point marker in the transaction command sequence

Syntax



Common Usage

```
SAVEPOINT savepoint_name;
```

Description

The SAVEPOINT command creates a save-point marker in the transaction log. If there is no active transaction in progress, the save-point will be marked and an implicit BEGIN DEFERRED TRANSACTION will be issued.

Save-points allow subsections of a transaction to be rewound and partially rolled back without losing the entire transaction state. A transaction can have multiple active save-points. Conceptually, these save-points act as if they were on a stack. Save-point names do not need to be unique.

Save-points are useful in multistep processes where each step needs to be attempted and verified before proceeding to the next step. By creating save-points before starting each step, it may be possible to revert just a single step, rather than the whole transaction, when a logical error case is encountered.

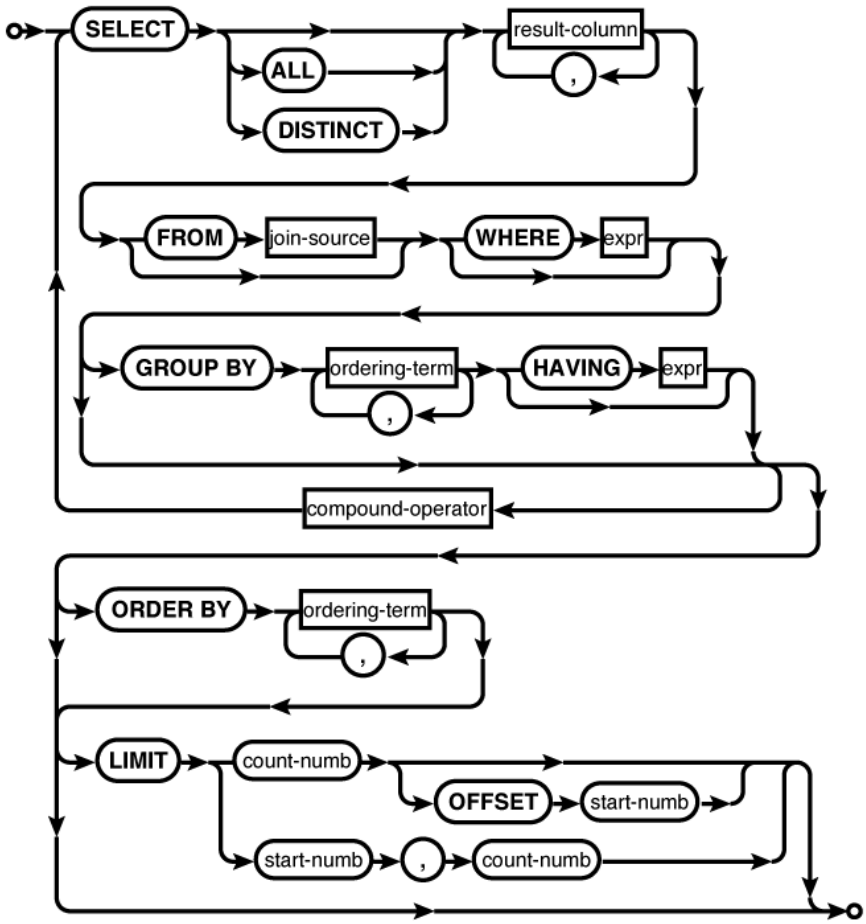
See Also

RELEASE SAVEPOINT, ROLLBACK TRANSACTION, BEGIN TRANSACTION

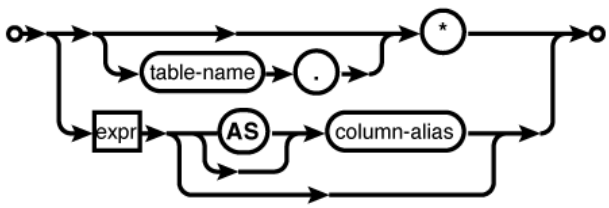
SELECT

Query data from the database

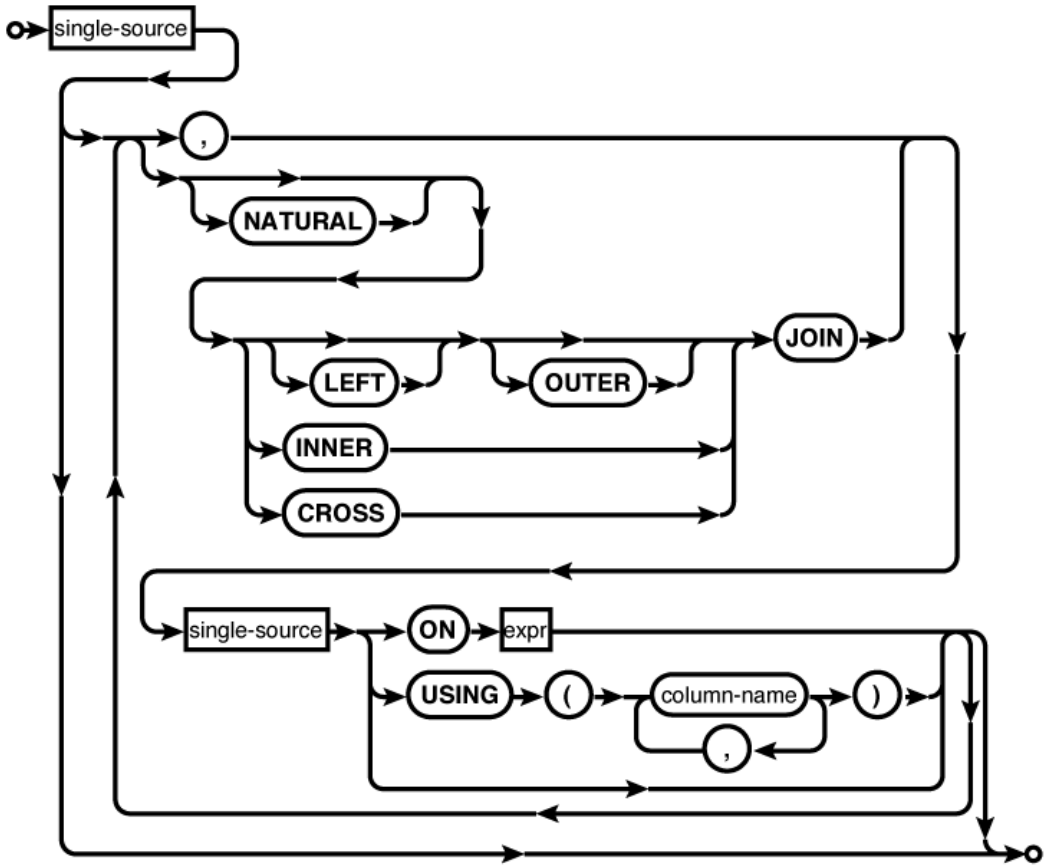
Syntax



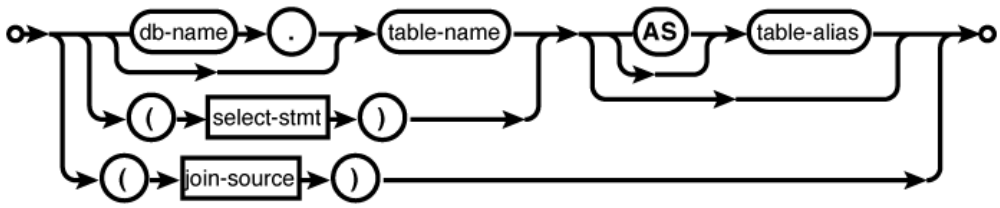
result-column:



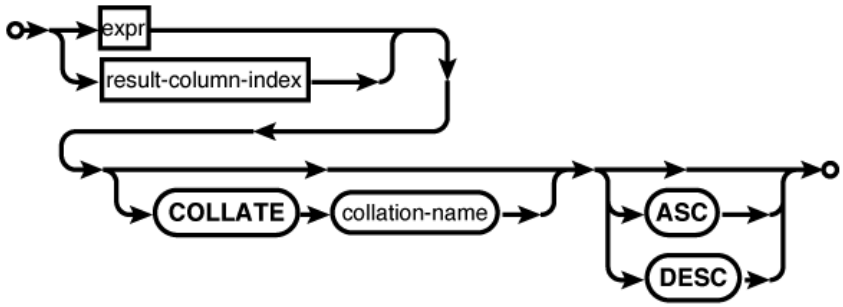
join-source:



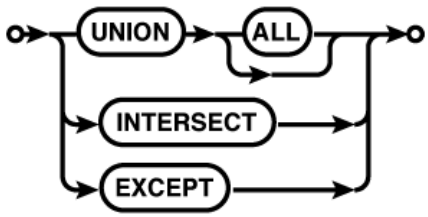
single-source:



ordering-term:



compound-operator:



Common Usage

```
SELECT * FROM tbl;
SELECT name FROM employees WHERE employee_id = 54923;
SELECT 5 + 6;
```

Description

The `SELECT` command is used to query the database and return a result. The `SELECT` command is the only SQL command capable of returning a user-generated result, be it a table query or a simple expression. Most consider `SELECT` to be the most complex SQL command. Although the basic format is fairly easy to understand, it does take some experience to understand its full power.

All of Chapter 5 is devoted to the `SELECT` command.

Basic format

The core `SELECT` command follows a simple pattern that can be roughly described as `SELECT` *`output`* `FROM` *`input`* `WHERE` *`filter`*. The output section describes the data that makes up the result set, the input section describes what tables, views, subqueries, etc., will act as data sources, and the filter section sets up conditions on which rows are part of the result set and which rows are filtered out.

A `SELECT` can either be `SELECT ALL` (default) or `SELECT DISTINCT`. The `ALL` keyword returns all rows in the result set, regardless of their composition. The `DISTINCT` keyword will force the select statement to eliminate duplicate results in the result set. There is usually a considerable performance penalty for calculating larger `DISTINCT` results.

The result set columns are defined by a series of comma-separated expressions. Every SELECT statement must have at least one result expression. These expressions often consist of only a source column name, although they can be any general expression. The character * means "return all columns," and will include all standard table columns from all of the source tables. All the standard columns of a specific source table can be returned with the format *table_name*.*. In both cases, the ROWID column will not be included, although ROWID alias columns will be included. Virtual tables can also mark some columns as hidden. Like the ROWID column, hidden columns will not be returned by default, but can be explicitly named as a result column.

Result columns can be given explicit names with an optional AS clause (the actual AS keyword is optional as well). Unless an AS clause is given, the name of the output column is at the discretion of the database engine. If an application depends on matching the names of specific output columns, the columns should be given explicit names with an AS clause.

The FROM clause defines where the data comes from and how it is shuffled together. If no FROM clause is given, the SELECT statement will return only one row. Each source is joined together with a comma or a JOIN operation. The comma acts as an unconditional CROSS JOIN. Different sources, including tables, subqueries, or other JOIN statements, can be grouped together into a large transitory table, which is fed through the rest of the SELECT statement, and ultimately used to produce the result set. For more information on the specific JOIN operators, see "FROM Clause" on page 63.

Each data source, be it a named table or a subquery, can be given an optional AS clause. Similar to result set columns, the actual AS keyword is optional. The AS clause allows an alias to be assigned to a given source. This is important to disambiguate table instances (for example, in a self-join).

The WHERE clause is used to filter rows. Conceptually, the FROM clause, complete with joins, is used to define a large table that consists of every possible row combination. The WHERE clause is evaluated against each of those rows, passing only those rows that evaluate to true. The WHERE clause can also be used to define join conditions, by effectively having the FROM clause produce the Cartesian product of the two tables, and use the WHERE clause to filter out only those rows that meet the join condition.

Additional clauses

Beyond SELECT, FROM, and WHERE, the SELECT statement can do additional processing with GROUP BY, HAVING, ORDER BY, and LIMIT.

The GROUP BY clause allows sets of rows in the result set to be collapsed into single rows. Groups of rows that share equivalent values in all of the expressions listed in the GROUP BY clause will be condensed to a single row. Normally, every source column reference in the result set expressions should be a column or expression included in the GROUP BY clause, or the column should appear as a parameter of an aggregate function. The value of any other source column is the value of the last row in the group to be processed, effectively making the result undefined. If a GROUP BY expression is a literal integer, it is assumed to be a column index to the result set. For example, the clause GROUP BY 2 would group the result set using the values in the second result column.

A HAVING clause can only be used in conjunction with a GROUP BY clause. Like the WHERE clause, a HAVING expression is used as a row filter. The key difference is that the HAVING expression is applied after any GROUP BY manipulation. This sequence allows the HAVING expression to filter aggregate outputs. Be aware that the WHERE clause is usually more efficient, since it can eliminate rows earlier in the SELECT pipeline. If possible, filtering should be done in the WHERE clause, saving the HAVING clause to filter aggregate results.

The ORDER BY clause sorts the result set into a specific order. Typically, the output ordering is not defined. Rows are returned as they become available, and no attempt is made to return the results in any specific order. The ORDER BY clause can be used to enforce a specific output ordering. Output is sorted by each expression in the clause, in turn, from most specific to least specific. The fact that the output of a SELECT can be ordered is one of the key differences between an SQL table and a result set. As with GROUP BY, if one of the ORDER BY expressions is a literal integer, it is assumed to be a column index to the result set.

Finally, the LIMIT clause can be used to control how many rows are returned, starting at a specific offset. If no offset is provided, the LIMIT will start from the beginning of the result set. Note that the two syntax variations (comma or OFFSET) provide the parameters in the opposite order.

Since the row order of a result is undefined, a LIMIT is most frequently used in conjunction with an ORDER BY clause. Although it is not strictly required, including an ORDER BY brings some meaning to the limit and offset values. There are very few cases when it makes sense to use a LIMIT without some type of imposed ordering.

Compound statements

Compound statements allow one or more SELECT...FROM...WHERE...GROUP BY...HAVING substatements to be brought together using set operations. SQLite supports the UNION, UNION ALL, INTERSECT, and EXCEPT compound operators. Each SELECT statement in a compound SELECT must return the same number of columns. The names of the result set columns will be taken from the first SELECT statement.

The UNION operator returns the union of the SELECT statements. By default, the UNION operator is a proper set operator and will only return distinct rows (including those from a single table) . UNION ALL, by contrast, will return the full set of rows returned by each SELECT. The UNION ALL operator is significantly less expensive than the UNION operator, so the use of UNION ALL is encouraged, when possible.

The INTERSECT command will return the set of rows that appear in both SELECT statements. Like UNION, the INTERSECT operator is a proper set operation and will only return one instance of each unique row, no matter how many times that row appears in both result sets of the individual SELECT statements.

The EXCEPT compound operator acts as a set-wise subtraction operator. All unique rows in the first SELECT that do not appear in the second SELECT will be returned.

See Also

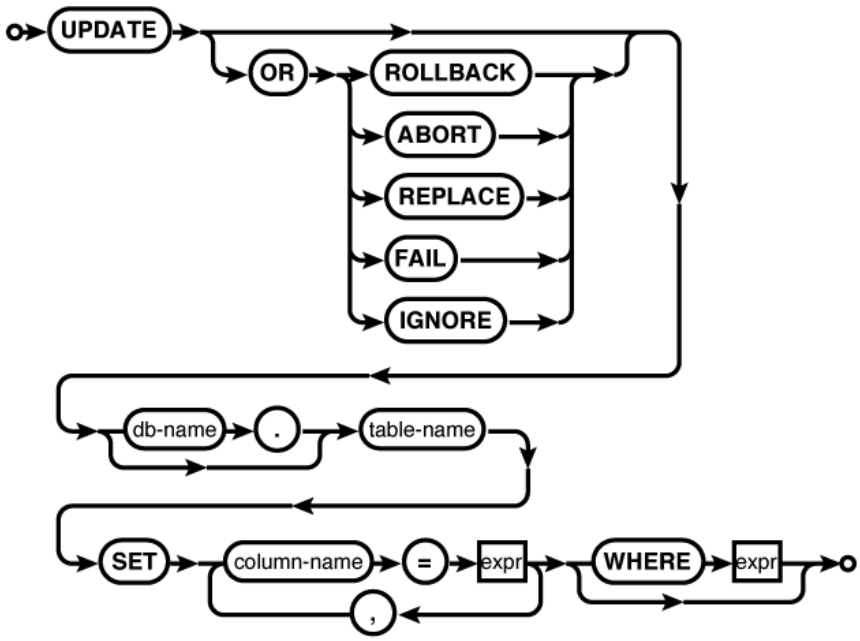
CREATE TABLE, INSERT, UPDATE, DELETE

Appendix C. SQLite SQL Command Reference. Using SQLite, ISBN: 9781449394592

UPDATE

Modify existing rows in a table

Syntax

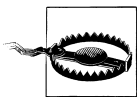


Common Usage

```
UPDATE database_name.table_name SET col5 = val5, col2 = val2 WHERE id = 42;
```

Description

The `UPDATE` command modifies one or more column values within existing table rows. The command starts out with a conflict resolution clause, followed by the name of the table that contains the rows we're updating. The table name is followed by a list of column names and new values. The final `WHERE` clause determines which rows are updated. A `WHERE` condition that causes no rows to be updated is not considered an error. If no `WHERE` clause is given, every row in the table is updated.



An `UPDATE` command with no `WHERE` clause will update *every* row in a table.

The values that are used to update a row may be given as expressions. These expressions are evaluated within the context of the original row values. This allows the value expression to refer to the old row value. For example, to increment a column value by one, you might find SQL similar to `UPDATE...SET col = col + 1 WHERE...`. Columns and values can be given in any order, as long as they appear in pairs. Any column, including `ROWID`, may be updated. Any columns that do not appear in the `UPDATE` command remain unmodified. There is no way to return a column to its default value.

If the SQLite library has been compiled with the optional SQLITE_ENABLE_UPDATE_DELETE_LIMIT directive, an optional ORDER BY...LIMIT clause may be used to update a specific number of rows. See the SQLite website (*http://www.sqlite.org/lang_update.html*) for more details.

The optional conflict resolution clause found at the beginning of the UPDATE (or INSERT) command is a nonstandard extension that controls how SQLite reacts to a constraint violation. For example, if a column must be unique, any attempt to update the value of that column to a value that is already in use by another row would cause a constraint violation. The constraint resolution clause determines how this situation is resolved.

ROLLBACK
: A ROLLBACK is immediately issued, rolling back any current transaction. An SQLITE_CONSTRAINT error is returned to the calling process. If no explicit transaction is currently in progress, the behavior will be identical to an ABORT.

ABORT
: This is the default behavior. Changes caused by the current command are undone and SQLITE_CONSTRAINT is returned, but no ROLLBACK is issued. For example, if a constraint violation is encountered on the fourth of ten possible row updates, the first three rows will be reverted, but the current transaction will remain active.

FAIL
: The command will fail and return SQLITE_CONSTRAINT, but any rows that have been previously modified will not be reverted. For example, if a constraint violation is encountered on the fourth of ten possible row updates, the first three modifications will be left in place and further processing will be cut short. The current transaction will not be committed or rolled back.

IGNORE
: Any constraint violation error is ignored. The row update will not be processed, but no error is returned. For example, if a constraint violation is encountered on the fourth of ten possible rows, not only are the first three row modifications left in place, processing continues with the remaining rows.

REPLACE
: The specific action taken by a REPLACE resolution depends on what type of constraint is violated.

  If a UNIQUE constraint is violated, the existing rows that are causing the constraint violation will first be deleted, and then the UPDATE (or INSERT) is allowed to be processed. No error is returned. This allows the command to succeed, but may result in one or more rows being deleted. In this case, any delete triggers associated with this table will not fire unless recursive triggers are enabled. Currently, the update hook is not called for automatically deleted rows, nor is the change counter incremented. These two behaviors may change in a future version, however.

  If a NOT NULL constraint is violated, the NULL is replaced by the default value for that column. If no default value has been defined, the ABORT resolution is used.

  If a CHECK constraint is violated, the IGNORE resolution is used.

Appendix C. SQLite SQL Command Reference. Using SQLite, ISBN: 9781449394592

Any conflict resolution clause found in an UPDATE (or INSERT) command will override any conflict resolution clause found in a table definition.

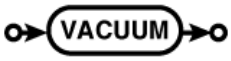
See Also

INSERT, DELETE, CREATE TABLE

VACUUM

Recover free space and optimize database

Syntax



Common Usage

```
VACUUM;
```

Description

The VACUUM command recovers free space from the database file and releases it to the filesystem. VACUUM can also defragment database structures and repack individual database pages. VACUUM can only be run against the main database (the database used to create the database connection). VACUUM has no effect on in-memory databases.

When data objects (rows, whole tables, indexes, etc.) are deleted or dropped from a database, the file size remains unchanged. Any database pages that are recovered from deleted objects are simply marked as free and available for any future database storage needs. As a result, under normal operations the database file can only grow in size.

Additionally, as rows are inserted and deleted from the database, the tables and indexes can become fragmented. In a dynamic database that normally experiences a high number of inserts, updates, and deletes, it is common for free pages to be scattered all across the database file. If a table or index requires additional pages for more storage, these will first be allocated off the free list. This means the actual parts of the database file that hold a particular table or index may become scattered and mixed all across the database file, lowering seek performance.

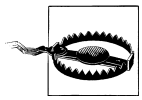
Finally, as rows are inserted, updated, and deleted, unused data blocks and other "holes" may appear within the individual database pages. This reduces the number of records that can be stored in a single page, increasing the total number of pages required to hold a table. In effect, this increases the storage overhead for the table, increasing read/write times and decreasing cache performance.

The vacuum process addresses all three of these issues by copying all the data within a database file to a separate, temporary database. This data transfer is done at a fairly high level, dealing with the logical elements of the database. As a result, individual database pages are "repacked," data objects are defragmented, and free pages are ignored. This optimizes storage space, reduces seek times, and recovers any free space from the database file. Once all this is done, the content of the temporary database file is copied back to the original file.

Appendix C. SQLite SQL Command Reference. Using SQLite, ISBN: 9781449394592

As the VACUUM command rebuilds the database file from scratch, VACUUM can also be used to modify many database-specific configuration parameters. For example, you can adjust the page size, file format, default encoding, and a number of other parameters that normally become fixed once a database file is created. To change something, just set the default new database pragma values to whatever you wish, and vacuum the database.

Be warned that this behavior is not always desirable. For example, if you have a database with a nondefault page size or file format, you need to be sure that you explicitly set all the correct pragma values before you vacuum the database. If you fail to do this, the database will be re-created with the default configuration values, rather than the original values. If you work with database files that have any nonstandard parameters, it is best to explicitly set all of these configuration values before you vacuum the database.



VACUUM will re-create the database using the current default values. For example, if you have a database that uses a custom page size and you wish to maintain that page size, you must issue the appropriate PRAGMA page_size command before issuing the VACUUM command. Failing to do so will result in the database being rebuilt with the default page size.

Logically, the database contents should remain unchanged from a VACUUM. The one exception is ROWID values. Columns marked INTEGER PRIMARY KEY will be preserved, but unaliased ROWID values may be reset. Also, indexes are rebuilt from scratch, rather than copied, so VACUUM does the equivalent of a REINDEX for each index in the database.

Generally, any reasonably dynamic database should vacuumed periodically. A good rule of thumb is to consider a full VACUUM any time 30% to 40% of the database content changes. It may also be helpful to VACUUM the database after a large table or index is dropped.

Be warned that the VACUUM process requires exclusive access to the database file and can take a significant amount of time to complete. VACUUM also requires enough disk space to hold the original file, plus the optimized copy of the database, plus a rollback journal that can be as large as the original file.

SQLite also supports an auto-vacuum mode, which enables portions of the vacuum process to be done automatically. It has some significant limitations, however, and even if auto-vacuum is enabled, it is still advisable to do a full manual VACUUM from time to time.

See Also

auto_vacuum [PRAGMA, Ap F], temp_store [PRAGMA, Ap F], temp_store_directory [PRAGMA, Ap F], REINDEX