SQLite Foreign Key Support

Table Of Contents

Overview

This document describes the support for SQL foreign key constraints introduced in SQLite version 3.6.19.

The first section introduces the concept of an SQL foreign key by example and defines the terminology used for the remainder of the document. Section 2 describes the steps an application must take in order to enable foreign key constraints in SQLite (it is disabled by default). The next section, section 3, describes the indexes that the user must create in order to use foreign key constraints, and those that should be created in order for foreign key constraints to function efficiently. Section 4 describes the advanced foreign key related features supported by SQLite and section 5 describes the way the ALTER and DROP TABLE commands are enhanced to support foreign key constraints. Finally, section 6 enumerates the missing features and limits of the current implementation.

This document does not contain a full description of the syntax used to create foreign key constraints in SQLite. This may be found as part of the documentation for the CREATE TABLE statement.

1. Introduction to Foreign Key Constraints

SQL foreign key constraints are used to enforce "exists" relationships between tables. For example, consider a database schema created using the following SQL commands:

```
CREATE TABLE artist(
  artistid    INTEGER PRIMARY KEY,
  artistname  TEXT
);
CREATE TABLE track(
  trackid     INTEGER,
  trackname   TEXT,
  trackartist INTEGER     -- Must map to an artist.artistid!
```

```
);
```

The applications using this database are entitled to assume that for each row in the *track* table there exists a corresponding row in the *artist* table. After all, the comment in the declaration says so. Unfortunately, if a user edits the database using an external tool or if there is a bug in an application, rows might be inserted into the *track* table that do not correspond to any row in the *artist* table. Or rows might be deleted from the *artist* table, leaving orphaned rows in the *track* table that do not correspond to any of the remaining rows in *artist*. This might cause the application or applications to malfunction later on, or at least make coding the application more difficult.

One solution is to add an SQL foreign key constraint to the database schema to enforce the relationship between the *artist* and *track* table. To do so, a foreign key definition may be added by modifying the declaration of the *track* table to the following:

```
CREATE TABLE track(
  trackid     INTEGER,
  trackname   TEXT,
  trackartist INTEGER,
  FOREIGN KEY(trackartist) REFERENCES artist(artistid)
);
```

This way, the constraint is enforced by SQLite. Attempting to insert a row into the *track* table that does not correspond to any row in the *artist* table will fail, as will attempting to delete a row from the *artist* table when there exist dependent rows in the *track* table There is one exception: if the foreign key column in the *track* table is NULL, then no corresponding entry in the *artist* table is required. Expressed in SQL, this means that for every row in the *track* table, the following expression evaluates to true:

```
trackartist IS NULL OR EXISTS(SELECT 1 FROM artist WHERE artistid=trackartist)
```

Tip: If the application requires a stricter relationship between *artist* and *track*, where NULL values are not permitted in the *trackartist* column, simply add the appropriate "NOT NULL" constraint to the schema.

There are several other ways to add an equivalent foreign key declaration to a CREATE TABLE statement. Refer to the CREATE TABLE documentation for details.

The following SQLite command-line session illustrates the effect of the foreign key constraint added to the *track* table:

```
sqlite> SELECT * FROM artist;
artistid  artistname
--------  -----------------
1         Dean Martin
2         Frank Sinatra

sqlite> SELECT * FROM track;
trackid  trackname          trackartist
-------  -----------------  -----------
11       That's Amore       1
12       Christmas Blues    1
13       My Way             2

sqlite> -- This fails because the value inserted into the trackartist column (3)
sqlite> -- does not correspond to row in the artist table.
```

```
sqlite> INSERT INTO track VALUES(14, 'Mr. Bojangles', 3);
SQL error: foreign key constraint failed

sqlite> -- This succeeds because a NULL is inserted into trackartist. A
sqlite> -- corresponding row in the artist table is not required in this case.
sqlite> INSERT INTO track VALUES(14, 'Mr. Bojangles', NULL);

sqlite> -- Trying to modify the trackartist field of the record after it has
sqlite> -- been inserted does not work either, since the new value of trackartist (3)
sqlite> -- Still does not correspond to any row in the artist table.
sqlite> UPDATE track SET trackartist = 3 WHERE trackname = 'Mr. Bojangles';
SQL error: foreign key constraint failed

sqlite> -- Insert the required row into the artist table. It is then possible to
sqlite> -- update the inserted row to set trackartist to 3 (since a corresponding
sqlite> -- row in the artist table now exists).
sqlite> INSERT INTO artist VALUES(3, 'Sammy Davis Jr.');
sqlite> UPDATE track SET trackartist = 3 WHERE trackname = 'Mr. Bojangles';

sqlite> -- Now that "Sammy Davis Jr." (artistid = 3) has been added to the database,
sqlite> -- it is possible to INSERT new tracks using this artist without violating
sqlite> -- the foreign key constraint:
sqlite> INSERT INTO track VALUES(15, 'Boogie Woogie', 3);
```

As you would expect, it is not possible to manipulate the database to a state that violates the foreign key constraint by deleting or updating rows in the *artist* table either:

```
sqlite> -- Attempting to delete the artist record for "Frank Sinatra" fails, since
sqlite> -- the track table contains a row that refer to it.
sqlite> DELETE FROM artist WHERE artistname = 'Frank Sinatra';
SQL error: foreign key constraint failed

sqlite> -- Delete all the records from the track table that refer to the artist
sqlite> -- "Frank Sinatra". Only then is it possible to delete the artist.
sqlite> DELETE FROM track WHERE trackname = 'My Way';
sqlite> DELETE FROM artist WHERE artistname = 'Frank Sinatra';

sqlite> -- Try to update the artistid of a row in the artist table while there
sqlite> -- exists records in the track table that refer to it.
sqlite> UPDATE artist SET artistid=4 WHERE artistname = 'Dean Martin';
SQL error: foreign key constraint failed

sqlite> -- Once all the records that refer to a row in the artist table have
sqlite> -- been deleted, it is possible to modify the artistid of the row.
sqlite> DELETE FROM track WHERE trackname IN('That''s Amore', 'Christmas Blues');
sqlite> UPDATE artist SET artistid=4 WHERE artistname = 'Dean Martin';
```

SQLite uses the following terminology:

- The **parent table** is the table that a foreign key constraint refers to. The parent table in the example in this section is the *artist* table. Some books and articles refer to this as the *referenced table*, which is arguably more correct, but tends to lead to confusion.
- The **child table** is the table that a foreign key constraint is applied to and the table that contains the REFERENCES clause. The example in this section uses the *track* table as the child table. Other books and articles refer to this as the *referencing table*.
- The **parent key** is the column or set of columns in the parent table that the foreign key constraint refers to. This is normally, but not always, the primary key of the parent table. The parent key must be a named column or columns in the parent table, not the rowid.

- The **child key** is the column or set of columns in the child table that are constrained by the foreign key constraint and which hold the REFERENCES clause.

The foreign key constraint is satisfied if for each row in the child table either one or more of the child key columns are NULL, or there exists a row in the parent table for which each parent key column contains a value equal to the value in its associated child key column.

In the above paragraph, the term "equal" means equal when values are compared using the rules specified here. The following clarifications apply:

- When comparing text values, the collating sequence associated with the parent key column is always used.
- When comparing values, if the parent key column has an affinity, then that affinity is applied to the child key value before the comparison is performed.

2. Enabling Foreign Key Support

In order to use foreign key constraints in SQLite, the library must be compiled with neither SQLITE_OMIT_FOREIGN_KEY or SQLITE_OMIT_TRIGGER defined. If SQLITE_OMIT_TRIGGER is defined but SQLITE_OMIT_FOREIGN_KEY is not, then SQLite behaves as it did prior to version 3.6.19 - foreign key definitions are parsed and may be queried using PRAGMA foreign_key_list, but foreign key constraints are not enforced. The PRAGMA foreign_keys command is a no-op in this configuration. If OMIT_FOREIGN_KEY is defined, then foreign key definitions cannot even be parsed (attempting to specify a foreign key definition is a syntax error).

Assuming the library is compiled with foreign key constraints enabled, it must still be enabled by the application at runtime, using the PRAGMA foreign_keys command. For example:

```
sqlite> PRAGMA foreign_keys = ON;
```

Foreign key constraints are disabled by default (for backwards compatibility), so must be enabled separately for each database connection. (Note, however, that future releases of SQLite might change so that foreign key constraints enabled by default. Careful developers will not make any assumptions about whether or not foreign keys are enabled by default but will instead enable or disable them as necessary.) The application can also use a PRAGMA foreign_keys statement to determine if foreign keys are currently enabled. The following command-line session demonstrates this:

```
sqlite> PRAGMA foreign_keys;
0
sqlite> PRAGMA foreign_keys = ON;
sqlite> PRAGMA foreign_keys;
1
sqlite> PRAGMA foreign_keys = OFF;
sqlite> PRAGMA foreign_keys;
0
```

Tip: If the command "PRAGMA foreign_keys" returns no data instead of a single row containing "0" or "1", then the version of SQLite you are using does not support foreign keys (either because it is older than 3.6.19 or because it was compiled with SQLITE_OMIT_FOREIGN_KEY or SQLITE_OMIT_TRIGGER defined).

It is not possible to enable or disable foreign key constraints in the middle of a multi-

statement transaction (when SQLite is not in autocommit mode). Attempting to do so does not return an error; it simply has no effect.

3. Required and Suggested Database Indexes

Usually, the parent key of a foreign key constraint is the primary key of the parent table. If they are not the primary key, then the parent key columns must be collectively subject to a UNIQUE constraint or have a UNIQUE index. If the parent key columns have a UNIQUE index, then that index must use the collation sequences that are specified in the CREATE TABLE statement for the parent table. For example,

```
CREATE TABLE parent(a PRIMARY KEY, b UNIQUE, c, d, e, f);
CREATE UNIQUE INDEX i1 ON parent(c, d);
CREATE INDEX i2 ON parent(e);
CREATE UNIQUE INDEX i3 ON parent(f COLLATE nocase);

CREATE TABLE child1(f, g REFERENCES parent(a));                       -- Ok
CREATE TABLE child2(h, i REFERENCES parent(b));                       -- Ok
CREATE TABLE child3(j, k, FOREIGN KEY(j, k) REFERENCES parent(c, d)); -- Ok
CREATE TABLE child4(l, m REFERENCES parent(e));                       -- Error!
CREATE TABLE child5(n, o REFERENCES parent(f));                       -- Error!
CREATE TABLE child6(p, q, FOREIGN KEY(p, q) REFERENCES parent(b, c)); -- Error!
CREATE TABLE child7(r REFERENCES parent(c));                          -- Error!
```

The foreign key constraints created as part of tables *child1*, *child2* and *child3* are all fine. The foreign key declared as part of table *child4* is an error because even though the parent key column is indexed, the index is not UNIQUE. The foreign key for table *child5* is an error because even though the parent key column has a unique index, the index uses a different collating sequence. Tables *child6* and *child7* are incorrect because while both have UNIQUE indices on their parent keys, the keys are not an exact match to the columns of a single UNIQUE index.

If the database schema contains foreign key errors that require looking at more than one table definition to identify, then those errors are not detected when the tables are created. Instead, such errors prevent the application from preparing SQL statements that modify the content of the child or parent tables in ways that use the foreign keys. Errors reported when content is changed are "DML errors" and errors reported when the schema is changed are "DDL errors". So, in other words, misconfigured foreign key constraints that require looking at both the child and parent are DML errors. The English language error message for foreign key DML errors is usually "foreign key mismatch" but can also be "no such table" if the parent table does not exist. Foreign key DML errors are may be reported if:

- The parent table does not exist, or
- The parent key columns named in the foreign key constraint do not exist, or
- The parent key columns named in the foreign key constraint are not the primary key of the parent table and are not subject to a unique constraint using collating sequence specified in the CREATE TABLE, or
- The child table references the primary key of the parent without specifying the primary key columns and the number of primary key columns in the parent do not match the number of child key columns.

The last bullet above is illustrated by the following:

```
CREATE TABLE parent2(a, b, PRIMARY KEY(a,b));

CREATE TABLE child8(x, y, FOREIGN KEY(x,y) REFERENCES parent2);       -- Ok
CREATE TABLE child9(x REFERENCES parent2);                            -- Error!
```

```
CREATE TABLE child10(x,y,z, FOREIGN KEY(x,y,z) REFERENCES parent2);    -- Error!
```

By contrast, if foreign key errors can be recognized simply by looking at the definition of the child table and without having to consult the parent table definition, then the CREATE TABLE statement for the child table fails. Because the error occurs during a schema change, this is a DDL error. Foreign key DDL errors are reported regardless of whether or not foreign key constraints are enabled when the table is created.

Indices are not required for child key columns but they are almost always beneficial. Returning to the example in section 1, each time an application deletes a row from the *artist* table (the parent table), it performs the equivalent of the following SELECT statement to search for referencing rows in the *track* table (the child table).

```
SELECT rowid FROM track WHERE trackartist = ?
```

where ? in the above is replaced with the value of the *artistid* column of the record being deleted from the *artist* table (recall that the *trackartist* column is the child key and the *artistid* column is the parent key). Or, more generally:

```
SELECT rowid FROM <child-table> WHERE <child-key> = :parent_key_value
```

If this SELECT returns any rows at all, then SQLite concludes that deleting the row from the parent table would violate the foreign key constraint and returns an error. Similar queries may be run if the content of the parent key is modified or a new row is inserted into the parent table. If these queries cannot use an index, they are forced to do a linear scan of the entire child table. In a non-trivial database, this may be prohibitively expensive.

So, in most real systems, an index should be created on the child key columns of each foreign key constraint. The child key index does not have to be (and usually will not be) a UNIQUE index. Returning again to the example in section 1, the complete database schema for efficient implementation of the foreign key constraint might be:

```
CREATE TABLE artist(
  artistid    INTEGER PRIMARY KEY,
  artistname  TEXT
);
CREATE TABLE track(
  trackid     INTEGER,
  trackname   TEXT,
  trackartist INTEGER REFERENCES artist
);
CREATE INDEX trackindex ON track(trackartist);
```

The block above uses a shorthand form to create the foreign key constraint. Attaching a "REFERENCES <*parent-table*>" clause to a column definition creates a foreign key constraint that maps the column to the primary key of <*parent-table*>. Refer to the CREATE TABLE documentation for further details.

4. Advanced Foreign Key Constraint Features

4.1. Composite Foreign Key Constraints

A composite foreign key constraint is one where the child and parent keys are both composite keys. For example, consider the following database schema:

```
CREATE TABLE album(
  albumartist TEXT,
  albumname TEXT,
  albumcover BINARY,
  PRIMARY KEY(albumartist, albumname)
);

CREATE TABLE song(
  songid     INTEGER,
  songartist TEXT,
  songalbum TEXT,
  songname   TEXT,
  FOREIGN KEY(songartist, songalbum) REFERENCES album(albumartist, albumname)
);
```

In this system, each entry in the song table is required to map to an entry in the album table with the same combination of artist and album.

Parent and child keys must have the same cardinality. In SQLite, if any of the child key columns (in this case songartist and songalbum) are NULL, then there is no requirement for a corresponding row in the parent table.

4.2. Deferred Foreign Key Constraints

Each foreign key constraint in SQLite is classified as either immediate or deferred. Foreign key constraints are immediate by default. All the foreign key examples presented so far have been of immediate foreign key constraints.

If a statement modifies the contents of the database so that an immediate foreign key constraint is in violation at the conclusion the statement, an exception is thrown and the effects of the statement are reverted. By contrast, if a statement modifies the contents of the database such that a deferred foreign key constraint is violated, the violation is not reported immediately. Deferred foreign key constraints are not checked until the transaction tries to COMMIT. For as long as the user has an open transaction, the database is allowed to exist in a state that violates any number of deferred foreign key constraints. However, COMMIT will fail as long as foreign key constraints remain in violation.

If the current statement is not inside an explicit transaction (a BEGIN/COMMIT/ROLLBACK block), then an implicit transaction is committed as soon as the statement has finished executing. In this case deferred constraints behave the same as immediate constraints.

To mark a foreign key constraint as deferred, its declaration must include the following clause:

```
DEFERRABLE INITIALLY DEFERRED                -- A deferred foreign key constraint
```

The full syntax for specifying foreign key constraints is available as part of the CREATE TABLE documentation. Replacing the phrase above with any of the following creates an immediate foreign key constraint.

```
NOT DEFERRABLE INITIALLY DEFERRED            -- An immediate foreign key constraint
NOT DEFERRABLE INITIALLY IMMEDIATE           -- An immediate foreign key constraint
NOT DEFERRABLE                               -- An immediate foreign key constraint
```

```
DEFERRABLE INITIALLY IMMEDIATE              -- An immediate foreign key constraint
DEFERRABLE                                  -- An immediate foreign key constraint
```

The defer_foreign_keys pragma can be used to temporarily change all foreign key constraints to deferred regardless of how they are declared.

The following example illustrates the effect of using a deferred foreign key constraint.

```
-- Database schema. Both tables are initially empty.
CREATE TABLE artist(
  artistid    INTEGER PRIMARY KEY,
  artistname  TEXT
);
CREATE TABLE track(
  trackid     INTEGER,
  trackname   TEXT,
  trackartist INTEGER REFERENCES artist(artistid) DEFERRABLE INITIALLY DEFERRED
);

sqlite3> -- If the foreign key constraint were immediate, this INSERT would
sqlite3> -- cause an error (since as there is no row in table artist with
sqlite3> -- artistid=5). But as the constraint is deferred and there is an
sqlite3> -- open transaction, no error occurs.
sqlite3> BEGIN;
sqlite3>   INSERT INTO track VALUES(1, 'White Christmas', 5);

sqlite3> -- The following COMMIT fails, as the database is in a state that
sqlite3> -- does not satisfy the deferred foreign key constraint. The
sqlite3> -- transaction remains open.
sqlite3> COMMIT;
SQL error: foreign key constraint failed

sqlite3> -- After inserting a row into the artist table with artistid=5, the
sqlite3> -- deferred foreign key constraint is satisfied. It is then possible
sqlite3> -- to commit the transaction without error.
sqlite3>   INSERT INTO artist VALUES(5, 'Bing Crosby');
sqlite3> COMMIT;
```

A nested savepoint transaction may be RELEASEd while the database is in a state that does not satisfy a deferred foreign key constraint. A transaction savepoint (a non-nested savepoint that was opened while there was not currently an open transaction), on the other hand, is subject to the same restrictions as a COMMIT - attempting to RELEASE it while the database is in such a state will fail.

If a COMMIT statement (or the RELEASE of a transaction SAVEPOINT) fails because the database is currently in a state that violates a deferred foreign key constraint and there are currently nested savepoints, the nested savepoints remain open.

4.3. ON DELETE and ON UPDATE Actions

Foreign key ON DELETE and ON UPDATE clauses are used to configure actions that take place when deleting rows from the parent table (ON DELETE), or modifying the parent key values of existing rows (ON UPDATE). A single foreign key constraint may have different actions configured for ON DELETE and ON UPDATE. Foreign key actions are similar to triggers in many ways.

The ON DELETE and ON UPDATE action associated with each foreign key in an SQLite database is one of "NO ACTION", "RESTRICT", "SET NULL", "SET DEFAULT" or "CASCADE". If an action is not explicitly specified, it defaults to "NO ACTION".

- **NO ACTION**: Configuring "NO ACTION" means just that: when a parent key is modified or deleted from the database, no special action is taken.

- **RESTRICT**: The "RESTRICT" action means that the application is prohibited from deleting (for ON DELETE RESTRICT) or modifying (for ON UPDATE RESTRICT) a parent key when there exists one or more child keys mapped to it. The difference between the effect of a RESTRICT action and normal foreign key constraint enforcement is that the RESTRICT action processing happens as soon as the field is updated - not at the end of the current statement as it would with an immediate constraint, or at the end of the current transaction as it would with a deferred constraint. Even if the foreign key constraint it is attached to is deferred, configuring a RESTRICT action causes SQLite to return an error immediately if a parent key with dependent child keys is deleted or modified.

- **SET NULL**: If the configured action is "SET NULL", then when a parent key is deleted (for ON DELETE SET NULL) or modified (for ON UPDATE SET NULL), the child key columns of all rows in the child table that mapped to the parent key are set to contain SQL NULL values.

- **SET DEFAULT**: The "SET DEFAULT" actions are similar to "SET NULL", except that each of the child key columns is set to contain the columns default value instead of NULL. Refer to the CREATE TABLE documentation for details on how default values are assigned to table columns.

- **CASCADE**: A "CASCADE" action propagates the delete or update operation on the parent key to each dependent child key. For an "ON DELETE CASCADE" action, this means that each row in the child table that was associated with the deleted parent row is also deleted. For an "ON UPDATE CASCADE" action, it means that the values stored in each dependent child key are modified to match the new parent key values.

For example, adding an "ON UPDATE CASCADE" clause to the foreign key as shown below enhances the example schema from section 1 to allow the user to update the artistid (the parent key of the foreign key constraint) column without breaking referential integrity:

```
-- Database schema
CREATE TABLE artist(
  artistid    INTEGER PRIMARY KEY,
  artistname  TEXT
);
CREATE TABLE track(
  trackid     INTEGER,
  trackname   TEXT,
  trackartist INTEGER REFERENCES artist(artistid) ON UPDATE CASCADE
);

sqlite> SELECT * FROM artist;
artistid  artistname
--------  -----------------
1         Dean Martin
2         Frank Sinatra

sqlite> SELECT * FROM track;
trackid  trackname          trackartist
-------  -----------------  -----------
11       That's Amore       1
12       Christmas Blues    1
13       My Way             2

sqlite> -- Update the artistid column of the artist record for "Dean Martin".
sqlite> -- Normally, this would raise a constraint, as it would orphan the two
sqlite> -- dependent records in the track table. However, the ON UPDATE CASCADE clause
sqlite> -- attached to the foreign key definition causes the update to "cascade"
```

```
sqlite> -- to the child table, preventing the foreign key constraint violation.
sqlite> UPDATE artist SET artistid = 100 WHERE artistname = 'Dean Martin';

sqlite> SELECT * FROM artist;
artistid  artistname
--------  -----------------
2         Frank Sinatra
100       Dean Martin

sqlite> SELECT * FROM track;
trackid  trackname          trackartist
-------  -----------------  -----------
11       That's Amore       100
12       Christmas Blues    100
13       My Way             2
```

Configuring an ON UPDATE or ON DELETE action does not mean that the foreign key constraint does not need to be satisfied. For example, if an "ON DELETE SET DEFAULT" action is configured, but there is no row in the parent table that corresponds to the default values of the child key columns, deleting a parent key while dependent child keys exist still causes a foreign key violation. For example:

```
-- Database schema
CREATE TABLE artist(
  artistid    INTEGER PRIMARY KEY,
  artistname  TEXT
);
CREATE TABLE track(
  trackid     INTEGER,
  trackname   TEXT,
  trackartist INTEGER DEFAULT 0 REFERENCES artist(artistid) ON DELETE SET DEFAULT
);

sqlite> SELECT * FROM artist;
artistid  artistname
--------  -----------------
3         Sammy Davis Jr.

sqlite> SELECT * FROM track;
trackid  trackname          trackartist
-------  -----------------  -----------
14       Mr. Bojangles      3

sqlite> -- Deleting the row from the parent table causes the child key
sqlite> -- value of the dependent row to be set to integer value 0. However, this
sqlite> -- value does not correspond to any row in the parent table. Therefore
sqlite> -- the foreign key constraint is violated and an is exception thrown.
sqlite> DELETE FROM artist WHERE artistname = 'Sammy Davis Jr.';
SQL error: foreign key constraint failed

sqlite> -- This time, the value 0 does correspond to a parent table row. And
sqlite> -- so the DELETE statement does not violate the foreign key constraint
sqlite> -- and no exception is thrown.
sqlite> INSERT INTO artist VALUES(0, 'Unknown Artist');
sqlite> DELETE FROM artist WHERE artistname = 'Sammy Davis Jr.';

sqlite> SELECT * FROM artist;
artistid  artistname
--------  -----------------
0         Unknown Artist

sqlite> SELECT * FROM track;
trackid  trackname          trackartist
```

```
-------  -----------------  -----------
14       Mr. Bojangles      0
```

Those familiar with SQLite triggers will have noticed that the "ON DELETE SET DEFAULT" action demonstrated in the example above is similar in effect to the following AFTER DELETE trigger:

```
CREATE TRIGGER on_delete_set_default AFTER DELETE ON artist BEGIN
  UPDATE child SET trackartist = 0 WHERE trackartist = old.artistid;
END;
```

Whenever a row in the parent table of a foreign key constraint is deleted, or when the values stored in the parent key column or columns are modified, the logical sequence of events is:

1. Execute applicable BEFORE trigger programs,
2. Check local (non foreign key) constraints,
3. Update or delete the row in the parent table,
4. Perform any required foreign key actions,
5. Execute applicable AFTER trigger programs.

There is one important difference between ON UPDATE foreign key actions and SQL triggers. An ON UPDATE action is only taken if the values of the parent key are modified so that the new parent key values are not equal to the old. For example:

```
-- Database schema
CREATE TABLE parent(x PRIMARY KEY);
CREATE TABLE child(y REFERENCES parent ON UPDATE SET NULL);

sqlite> SELECT * FROM parent;
x
----
key

sqlite> SELECT * FROM child;
y
----
key

sqlite> -- Since the following UPDATE statement does not actually modify
sqlite> -- the parent key value, the ON UPDATE action is not performed and
sqlite> -- the child key value is not set to NULL.
sqlite> UPDATE parent SET x = 'key';
sqlite> SELECT IFNULL(y, 'null') FROM child;
y
----
key

sqlite> -- This time, since the UPDATE statement does modify the parent key
sqlite> -- value, the ON UPDATE action is performed and the child key is set
sqlite> -- to NULL.
sqlite> UPDATE parent SET x = 'key2';
sqlite> SELECT IFNULL(y, 'null') FROM child;
y
----
null
```

5. CREATE, ALTER and DROP TABLE commands

This section describes the way the CREATE TABLE, ALTER TABLE, and DROP TABLE commands interact with SQLite's foreign keys.

A CREATE TABLE command operates the same whether or not foreign key constraints are enabled. The parent key definitions of foreign key constraints are not checked when a table is created. There is nothing stopping the user from creating a foreign key definition that refers to a parent table that does not exist, or to parent key columns that do not exist or are not collectively bound by a PRIMARY KEY or UNIQUE constraint.

The ALTER TABLE command works differently in two respects when foreign key constraints are enabled:

- It is not possible to use the "ALTER TABLE ... ADD COLUMN" syntax to add a column that includes a REFERENCES clause, unless the default value of the new column is NULL. Attempting to do so returns an error.
- If an "ALTER TABLE ... RENAME TO" command is used to rename a table that is the parent table of one or more foreign key constraints, the definitions of the foreign key constraints are modified to refer to the parent table by its new name. The text of the child CREATE TABLE statement or statements stored in the sqlite_master table are modified to reflect the new parent table name.

If foreign key constraints are enabled when it is prepared, the DROP TABLE command performs an implicit DELETE to remove all rows from the table before dropping it. The implicit DELETE does not cause any SQL triggers to fire, but may invoke foreign key actions or constraint violations. If an immediate foreign key constraint is violated, the DROP TABLE statement fails and the table is not dropped. If a deferred foreign key constraint is violated, then an error is reported when the user attempts to commit the transaction if the foreign key constraint violations still exist at that point. Any "foreign key mismatch" errors encountered as part of an implicit DELETE are ignored.

The intent of these enhancements to the ALTER TABLE and DROP TABLE commands is to ensure that they cannot be used to create a database that contains foreign key violations, at least while foreign key constraints are enabled. There is one exception to this rule though. If a parent key is not subject to a PRIMARY KEY or UNIQUE constraint created as part of the parent table definition, but is subject to a UNIQUE constraint by virtue of an index created using the CREATE INDEX command, then the child table may be populated without causing a "foreign key mismatch" error. If the UNIQUE index is dropped from the database schema, then the parent table itself is dropped, no error will be reported. However the database may be left in a state where the child table of the foreign key constraint contains rows that do not refer to any parent table row. This case can be avoided if all parent keys in the database schema are constrained by PRIMARY KEY or UNIQUE constraints added as part of the parent table definition, not by external UNIQUE indexes.

The properties of the DROP TABLE and ALTER TABLE commands described above only apply if foreign keys are enabled. If the user considers them undesirable, then the workaround is to use PRAGMA foreign_keys to disable foreign key constraints before executing the DROP or ALTER TABLE command. Of course, while foreign key constraints are disabled, there is nothing to stop the user from violating foreign key constraints and thus creating an internally inconsistent database.

6. Limits and Unsupported Features

This section lists a few limitations and omitted features that are not mentioned elsewhere.

1. **No support for the MATCH clause.** According to SQL92, a MATCH clause may be

attached to a composite foreign key definition to modify the way NULL values that occur in child keys are handled. If "MATCH SIMPLE" is specified, then a child key is not required to correspond to any row of the parent table if one or more of the child key values are NULL. If "MATCH FULL" is specified, then if any of the child key values is NULL, no corresponding row in the parent table is required, but all child key values must be NULL. Finally, if the foreign key constraint is declared as "MATCH PARTIAL" and one of the child key values is NULL, there must exist at least one row in the parent table for which the non-NULL child key values match the parent key values.

SQLite parses MATCH clauses (i.e. does not report a syntax error if you specify one), but does not enforce them. All foreign key constraints in SQLite are handled as if MATCH SIMPLE were specified.

2. **No support for switching constraints between deferred and immediate mode.** Many systems allow the user to toggle individual foreign key constraints between deferred and immediate mode at runtime (for example using the Oracle "SET CONSTRAINT" command). SQLite does not support this. In SQLite, a foreign key constraint is permanently marked as deferred or immediate when it is created.

3. **Recursion limit on foreign key actions.** The SQLITE_MAX_TRIGGER_DEPTH and SQLITE_LIMIT_TRIGGER_DEPTH settings determine the maximum allowable depth of trigger program recursion. For the purposes of these limits, foreign key actions are considered trigger programs. The PRAGMA recursive_triggers setting does not affect the operation of foreign key actions. It is not possible to disable recursive foreign key actions.