3

SQLite Syntax and Use

IN THIS CHAPTER WE LOOK IN DETAIL at the SQL syntax understood by SQLite. We will discuss the full capabilities of the language and you will learn to write effective, accurate SQL.

You have already come across most of the supported SQL commands in Chapter 2, "Working with Data," in the context of the demo database. This chapter builds on that knowledge by exploring the syntax and usage of each command in more detail to give a very broad overview of what you can do using SQLite.

Naming Conventions

Each database, table, column, index, trigger, or view has a name by which it is identified and almost always the name is supplied by the developer. The rules governing how a valid identifier is formed in SQLite are set out in the next few sections.

Valid Characters

An identifier name must begin with a letter or the underscore character, which may be followed by a number of alphanumeric characters or underscores. No other characters may be present. These identifier names are valid:

- `mytable`
- `my_field`
- `xyz123`
- `a`

However, the following are not valid identifiers:

- `my table`
- `my-field`
- `123xyz`

You can use other characters in identifiers if they are enclosed in double quotes (or square brackets), for example:

```
sqlite> CREATE TABLE "123 456"("hello-world", " ");
```

Name Length

SQLite does not have a fixed upper limit on the length of an identifier name, so any name that you find manageable to work with is suitable.

Reserved Keywords

Care must be taken when using SQLite keywords as identifier names. As a general rule of thumb you should try to avoid using any keywords from the SQL language as identifiers, although if you really want to do so, they can be used providing they are enclosed in square brackets.

For instance the following statement will work just fine, but this should not be mimicked on a real database for the sake of your own sanity.

```
sqlite> CREATE TABLE [TABLE] (
   ...>   [SELECT],
   ...>   [INTEGER] INTEGER,
   ...>   [FROM],
   ...>   [TABLE]
   ...> );
```

Case Sensitivity

For the most part, case sensitivity in SQLite is off. Table names and column names can be typed in uppercase, lowercase, or mixed case, and different capitalizations of the same database object name can be used interchangeably.

SQL commands are always shown in this book with the keywords in uppercase for clarity; however, this is not a requirement.

> **Note**
>
> The `CREATE TABLE`, `CREATE VIEW`, `CREATE INDEX`, and `CREATE TRIGGER` statements all store the exact way in which they were typed to the database so that the command used to create a database object can be retrieved by querying the `sqlite_master` table. Therefore it is always a good idea to format your `CREATE` statements clearly, so they can be referred to easily in the future.

Creating and Dropping Tables

Creating and dropping database tables in SQLite is performed with the `CREATE TABLE` and `DROP TABLE` commands respectively. The basic syntax for `CREATE TABLE` is as follows:

```
CREATE [TEMP | TEMPORARY] TABLE table-name (
  column-def[, column-def]*
  [,constraint]*
);
```

Simply put, a table may be declared as temporary, if desired, and the structure of each table has to have one or more column definitions followed by zero or more constraints.

Table Column Definitions

A column definition is defined as follows:

```
name [type] [[CONSTRAINT name] column-constraint]*
```

As you saw in Chapter 2, SQLite is typeless and therefore the `type` attribute is actually optional. Except for an `INTEGER PRIMARY KEY` column, the data type is only used to determine whether values stored in that column are to be treated as strings or numbers when compared to other values.

You can use the optional `CONSTRAINT` clause to specify one or more of the following column constraints that should be enforced when data is inserted:

- `NOT NULL`
- `DEFAULT`
- `PRIMARY KEY`
- `UNIQUE`

A column declared as `NOT NULL` must contain a value; otherwise, an `INSERT` attempt will fail, as demonstrated in the following example:

```
sqlite> CREATE TABLE vegetables (
   ...>   name CHAR NOT NULL,
   ...>   color CHAR NOT NULL
   ...> );

sqlite> INSERT INTO vegetables (name) VALUES ('potato');
SQL error: vegetables.color may not be NULL
```

Often, a column declared `NOT NULL` is also given a `DEFAULT` value, which will be used automatically if that column is not specified in an `INSERT`. The following example shows this in action.

```
sqlite> CREATE TABLE vegetables (
   ...>   name CHAR NOT NULL,
   ...>   color CHAR NOT NULL DEFAULT 'green’
   ...> );

sqlite> INSERT INTO vegetables (name, color) VALUES ('carrot’, 'orange’);
sqlite> INSERT INTO vegetables (name) VALUES ('bean’);
sqlite> SELECT * FROM vegetables;
name        color
----------  ----------
carrot      orange
bean        green
```

However, if you attempt to insert `NULL` explicitly into a `NOT NULL` column, SQLite will still give an error:

```
sqlite> INSERT INTO vegetables (name, color) VALUES ('cabbage’, NULL);
SQL error: vegetables.color may not be NULL
```

Functionally, a `PRIMARY KEY` column behaves just the same as one with a `UNIQUE` constraint. Both types of constraint enforce that the same value may only be stored in that column once, but other than the special case of an `INTEGER PRIMARY KEY`, the only point to note is that a table can have only one `PRIMARY KEY` column.

SQLite will raise an error whenever an attempt is made to insert a duplicate value into a `UNIQUE` or `PRIMARY KEY` column, as shown in the following example. This example also shows that a column can be declared as both `NOT NULL` and `UNIQUE`.

```
sqlite> CREATE TABLE vegetables (
   ...>   name CHAR NOT NULL UNIQUE,
   ...>   color CHAR NOT NULL
   ...> );
sqlite> INSERT INTO vegetables (name, color) VALUES ('pepper’, 'red’);
sqlite> INSERT INTO vegetables (name, color) VALUES ('pepper’, 'green’);
SQL error: column name is not unique
```

Resolving Conflicts

`NOT NULL`, `PRIMARY KEY`, and `UNIQUE` constraints may all be used in conjunction with an `ON CONFLICT` clause to specify the way a conflict should be resolved if an attempt to insert or modify data violates a column constraint.

The conflict resolution algorithms supported are

- `ROLLBACK`
- `ABORT`
- `FAIL`
- `IGNORE`
- `REPLACE`

You could apply a constraint to the `vegetables` table from the preceding example as follows:

```
sqlite> CREATE TABLE vegetables (
   ...>   name CHAR NOT NULL UNIQUE ON CONFLICT REPLACE,
   ...>   color CHAR NOT NULL
   ...> );
```

This time, because REPLACE was specified as the conflict resolution algorithm, inserting the same vegetable name twice does not cause an error. Instead the new record replaces the conflicting record.

```
sqlite> INSERT INTO vegetables (name, color) VALUES ('pepper', 'red');
sqlite> INSERT INTO vegetables (name, color) VALUES ('pepper', 'green');
sqlite> SELECT * FROM vegetables;
name        color
----------  ----------
pepper      green
```

The REPLACE algorithm ensures that an SQL statement is always executed, even if a UNIQUE constraint would otherwise be violated. Before the UPDATE or INSERT takes place, any pre-existing rows that would cause the violation are removed. If a NOT NULL constraint is violated and there is no DEFAULT value, the ABORT algorithm is used instead.

The ROLLBACK algorithm causes an immediate ROLLBACK TRANSACTION to be issued as soon as the conflict occurs and the command will exit with an error.

When you use the ABORT algorithm, no ROLLBACK TRANSACTION is issued, so if the violation occurs within a transaction consisting of more than one INSERT or UPDATE, the database changes from the previous statements will remain. Any changes attempted by the statement causing the violation, however, will not take place. For a single command using only an implicit transaction, the behavior is identical to ROLLBACK.

The FAIL algorithm causes SQLite to stop with an error when a constraint is violated; however, any changes made as part of that command up to the point of failure will be preserved. For instance, when an UPDATE statement performs a change sequentially on many rows of the database, any rows affected before the constraint was violated will remain updated.

SQLite will never stop with an error when the IGNORE algorithm is specified and the constraint violation is simply passed by. In the case of an UPDATE affecting multiple rows, the modification will take place for every row other than the one that causes the conflict, both before and after.

The ON CONFLICT clause in a CREATE TABLE statement has the lowest precedence of all the places in which it can be specified. An overriding conflict resolution algorithm can be specified in the ON CONFLICT clause of a BEGIN TRANSACTION command, which can in turn be overridden by the OR clause of a COPY, INSERT, or UPDATE statement. We will see the respective syntaxes for these clauses later in this chapter.

The CHECK Clause

The CREATE TABLE syntax also allows for a CHECK clause to be defined, with an expression in parentheses. This is a feature included for SQL compatibility and is reserved for future use, but at the time of this writing is not implemented.