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.

Using Temporary Tables

Using `CREATE TEMPORARY TABLE` creates a table object in SQLite that can be queried and manipulated exactly the same as a nontemporary table. However, the table will only be visible to the process in which it was created and will be destroyed as soon as the database is closed.

```
$ sqlite tempdb
SQLite version 2.8.12
Enter ".help" for instructions
sqlite> CREATE TEMPORARY TABLE temptable (
   ...>   myfield char
   ...> );
sqlite> INSERT INTO temptable (myfield) VALUES ('abc');
sqlite> .quit

$ sqlite tempdb
SQLite version 2.8.12
Enter ".help" for instructions
sqlite> INSERT INTO temptable (myfield) VALUES ('xyz');
SQL error: no such table: temptable
```

The data inserted into a temporary table and its schema are not written to the connected database file, nor is there a record created in `sqlite_master`. Instead a separate table, `sqlite_temp_master`, is used to reference temporary tables.

```
sqlite> CREATE TEMPORARY TABLE temptable (
   ...>   myfield char
   ...> );
sqlite> SELECT * FROM sqlite_temp_master;
    type = table
    name = temptable
tbl_name = temptable
rootpage = 3
     sql = CREATE TEMPORARY TABLE temptable (
  myfield char
)
```

Temporary tables are specific to the `sqlite` handle, not the process. Surprisingly, people often become confused about this, particularly in Windows, where a common design pattern is to open a separate `sqlite` handle to the same database from each thread.

Anatomy of a SELECT Statement

The syntax definition for an SQL statement is

```
SELECT [ALL | DISTINCT] result [FROM table-list]
[WHERE expr]
[GROUP BY expr-list]
```

```
[HAVING expr]
[compound-op select]*
[ORDER BY sort-expr-list]
[LIMIT integer [(OFFSET|,) integer]]
```

The only required item in a `SELECT` statement is the `result`, which can be one of the following:

- The * character
- A comma-separated list of one or more column names
- An expression

The latter two bullet points should be combined into: "A comma-separated list of one or more expressions." The original two points make it seem as if the following would be an error:

```
SELECT a+1, b+1 FROM ab;
```

but this would be okay:

```
SELECT a, b FROM ab;
```

In fact, both are valid.

Using the * character or a list of columns makes no sense without a `FROM` clause, but in fact an expression whose arguments are constants rather than database items can be used alone in a `SELECT` statement, as in the following examples.

```
sqlite> SELECT (60 * 60 * 24);
86400

sqlite> SELECT max(5, 20, -4, 8.7);
20

sqlite> SELECT random();
220860261
```

If the `FROM` list is omitted, SQLite effectively evaluates the expression against a table that always contains a single row.

The `FROM` list includes one or more table names in a comma-separated list, each with an optional alias name that can be used to qualify individual column names in the `result`. Where aliases are not used, the table name in full can be used to qualify columns.

For instance, the two following `SELECT` statements are identical—the latter uses a table alias `m` for `mytable`.

```
SELECT mytable.myfield
FROM myfield;

SELECT m.myfield
FROM mytable m;
```

The WHERE Clause

The `WHERE` clause specifies one or more conditions used to impose restrictions on the dataset returned by a `SELECT`. It is used both to limit the number of rows returned and to indicate a relationship used to join two tables together.

The general usage to impose a condition on the rows in a table is as follows:

```
SELECT result
FROM table-list
WHERE expr;
```

Expression `expr` generally involves a comparison of some kind on a table column, as shown in the following example:

```
SELECT *
FROM mytable
WHERE myfield = 'somevalue';
```

Table 3.1 shows the relational operators that can be used in a `WHERE` clause.

Table 3.1 **Relational Operators**

Operator	Meaning
`a = b`	a is equal to b
`a != b`	a is not equal to b
`a < b`	a is less than b
`a > b`	a is greater than b
`a <= b`	a is less than or equal to b
`a >= b`	a is greater than or equal to b
`a IN (b, c)`	a is equal to either b or c
`a NOT IN (b, c)`	a is equal to neither b nor c

When you perform a comparison using a relational operator—and particularly the greater-than and less-than operators, `<` and `>`—the data type of the column comes into play.

The following example shows how comparisons between the numbers 8, 11, and 101 differ greatly when performed as string operations. As integers, the order is as you would expect, however as strings,‘`101'` is less than `'11'`, which is in turn less than `'8'`. The individual character values in the string are compared from left to right in turn to determine which is the greatest value.

```
sqlite> CREATE TABLE compare (string TEXT, number INTEGER);
sqlite> INSERT INTO compare (string, number) values ('101', 101);
sqlite> SELECT number FROM compare WHERE number > 11;
101
sqlite> SELECT string FROM compare WHERE string > '11';
8
```

However, note that if you use a relational operator with a number argument that is not contained in quotes, an integer comparison is performed regardless of the column data type.

```
sqlite> SELECT string FROM compare WHERE string > 11;
101
```

Selecting from multiple tables without a `WHERE` clause produces a Cartesian product of the datasets and is usually not a desirable result. With two tables, each record in `table1` is paired with each record in `table2`. The total number of rows returned is the product of the number of rows in each table in the `FROM` list.

The following example shows the result of a Cartesian product of three tables, each containing just two rows. In total, eight rows are returned (2 2 2).

```
sqlite> SELECT table1.myfield, table2.myfield, table3.myfield
   ...> FROM table1, table2, table3;
table1.myfield  table2.myfield  table3.myfield
--------------  --------------  --------------
Table 1 row 1   Table 2 row 1   Table 3 row 1
Table 1 row 1   Table 2 row 1   Table 3 row 2
Table 1 row 1   Table 2 row 2   Table 3 row 1
Table 1 row 1   Table 2 row 2   Table 3 row 2
Table 1 row 2   Table 2 row 1   Table 3 row 1
Table 1 row 2   Table 2 row 1   Table 3 row 2
Table 1 row 2   Table 2 row 2   Table 3 row 1
Table 1 row 2   Table 2 row 2   Table 3 row 2
```

In this example each table has a field called `myfield`, so each column in the `result` has to be qualified with the appropriate table name. This is not necessary where a column name is unique across all tables in the `FROM` list; however, it is good practice to always qualify column names to avoid ambiguity. If a column name could refer to more than one table, SQLite will not make the decision. Instead an error is raised as shown in the following example:

```
sqlite> SELECT myfield
   ...> FROM table1, table2, table3;
SQL error: ambiguous column name: myfield
```

To join two tables on a common field—known as an equi-join because the relationship is an equality—the general syntax is

```
SELECT result
FROM table1, table2
WHERE table1.keyfield1 = table2.keyfield2
```

SQLite supports outer joins via the `LEFT JOIN` keyword, whereby each row in the left table—the one specified first in the `SELECT` statement—is combined with a row from the right table. Where the join condition does not produce a match between the two tables, rows from the left table are still returned but with NULL values for each column that should be in the right table.

The general syntax for a LEFT JOIN is as follows:

```
SELECT result
FROM table1
LEFT [OUTER] JOIN table2
ON table1.keyfield1 = table2.keyfield2
[WHERE expr]
```

The LEFT JOIN operator can be written as LEFT OUTER JOIN as a matter of preference; the OUTER keyword is optional.

GROUP BY and Aggregate Functions

The GROUP BY clause is used to aggregate data into a single row where the value of one or more specified columns is repeated. This feature can be used to reduce the number of records to only find unique values of a column, but is particularly useful when used in conjunction with the SQLite's aggregate functions.

The GROUP BY clause takes a list of expressions—usually column names from the result—and aggregates data for each expression. In the vegetables table we created previously we had more than one green vegetable, so grouping on the color column will return each value only once.

```
sqlite> SELECT color
   ...> FROM vegetables
   ...> GROUP BY color;
color
----------
green
orange
```

More interesting is to use the aggregate function count() to show how many records there are for each value of color:

```
sqlite> SELECT color, count(color)
   ...> FROM vegetables
   ...> GROUP BY color;
color       count(color)
----------  ------------
green       2
orange      1
```

Using count(fieldname) will return the number of rows containing a non-NULL value in that field. If you want to return a count of the total number of rows, regardless of any NULL values, count(*) will do this, as the following example shows:

```
sqlite> CREATE TABLE mytable (
   ...>   field1 CHAR,
   ...>   field2 INTEGER
   ...> );

sqlite> INSERT INTO mytable VALUES ('foo', 5);
sqlite> INSERT INTO mytable VALUES ('foo', 14);
sqlite> INSERT INTO mytable VALUES ('bar', 25);
sqlite> INSERT INTO mytable VALUES ('bar', 8);
sqlite> INSERT INTO mytable VALUES ('bar', NULL);

sqlite> SELECT field1, count(field2), count(*)
   ...> FROM mytable
   ...> GROUP BY field1;
field1      count(field2)  count(*)
----------  -------------  ----------
bar         2              3
foo         2              2
```

There are also a number of aggregate functions for performing summary calculations on grouped data, as shown in the following example:

```
sqlite> SELECT field1, sum(field2), min(field2), max(field2), avg(field2)
   ...> FROM mytable
   ...> GROUP BY field1;
field1      sum(field2)  min(field2)  max(field2)  avg(field2)
----------  -----------  -----------  -----------  -----------
bar         33           8            25           16.5
foo         19           5            14           9.5
```

Table 3.2 lists all the aggregate functions available in SQLite.

Table 3.2 **Aggregate Functions**

Function	Meaning
`avg(column)`	Returns the mean average of all values in `column`
`count(column)`	Returns the number of times that a non-`NULL` value appears in `column`
`count(*)`	Returns the total number of rows in a query, regardless of `NULL` values
`max(column)`	Returns the highest of all values in `column`, using the usual sort order
`min(column)`	Returns the lowest of all values in `column`, using the usual sort order
`sum(column)`	Returns the numeric sum of all values in `column`

HAVING Clause

The `HAVING` clause is a further condition applied after aggregation takes place. In contrast to a `WHERE` clause, which applies a condition to individual elements in a table, `HAVING` is used to restrict records based on the summary value of a grouping.

To return only rows from the `vegetables` table where there is more than one of the same color, we can do this:

```
sqlite> SELECT color, count(*)
   ...> FROM vegetables GROUP BY color
   ...> HAVING count(*) > 1;
color       count(*)
----------  ----------
green       2
```

It is actually not necessary for `count(*)` to appear in the `result`, as shown in the following example:

```
sqlite> SELECT color
   ...> FROM vegetables GROUP BY color
   ...> HAVING count(*) > 1;
color
----------
green
```

Column Aliases

The column headings displayed in the output of a `SELECT` statement are usually the same as the items specified in the `result` section. For a straight column, the name of the column is displayed. For an expression, however, the expression text is used.

Although the column headings are only displayed in `sqlite` when `.headers` is set to `on`, it is important to know what each column's name is so that all the columns can be referenced correctly from within a programming API. A column alias is specified with the `AS` keyword to explicitly give a new name to a selected column.

In the following example, we give a column alias to the result of the `count(*)` aggregate function. In addition to renaming the column header, we can use the alias name in the `HAVING` clause, which can sometimes aid readability of code.

```
sqlite> SELECT color, count(*) AS num_colors
   ...> FROM vegetables GROUP BY color
   ...> HAVING num_colors > 1;
color       num_colors
----------  ----------
green       2
```

Attaching to Another Database

Using `sqlite`, the `.databases` command lists all the databases that are open for the current session. There will always be two databases open after you invoke `sqlite`—*main*, the database specified on the command line, and *temp*, the database used for temporary tables.

```
sqlite> .databases
0     main        /home/chris/sqlite/demodb
1     temp        /var/tmp/sqlite VGazbfyWvuUr29P
```

It is possible to attach more databases to your current session with the `ATTACH DATABASE` statement. This adds a connection to another database without replacing your currently selected database.

The syntax is

```
ATTACH [DATABASE] database-filename AS database-name
```

The keyword `DATABASE` is optional and is used only for readability, but you must provide a unique `database-name` parameter that will be used to qualify table references, essential in case more than one database could have the same table name.

Suppose you are working on a new database called `newdb` and want to access some of the databases from our demo database from Chapter 2. The following example shows `demodb` being attached to the current `sqlite` session:

```
$ sqlite newdb
SQLite version 2.8.12
Enter ".help" for instructions
sqlite> ATTACH DATABASE demodb AS demodb;
sqlite> .databases
0     main        /home/chris/sqlite/newdb
1     temp        /var/tmp/sqlite_VGazbfyWvuUr29P
2     demodb      /home/chris/sqlite/demodb
```

Accessing tables from an attached database is straightforward—just prefix any table name with the database name (the name given after the keyword `AS`, not the filename, if they are different) and a period.

We can perform a query on the `clients` table from `demodb` as follows:

```
sqlite> SELECT company_name FROM demodb.clients;
company_name
--------------------
Acme Products
ABC Enterprises
Premier Things Ltd
```

Tables in the `main` database can be accessed using their table name alone, or qualified as `main.tablename`. If a table name is unique across all databases attached in a particular session, it does not need to be prefixed with its database name even if it is not in the `main` database. However, it is still good practice to qualify all tables when you are working with multiple databases to avoid confusion.

Note

The SQL commands `INSERT`, `UPDATE`, `SELECT`, and `DELETE` can all be performed on an attached database by using the database name prefix. However, `CREATE TABLE` and `DROP TABLE` can only take place on the `main` database—you must exit `sqlite` and begin a new session if you want to manipulate tables from a different database.

Note the situation with multi-database transactions here. If a machine or software failure occurs, a transaction is only atomic within one database. If more than one database were written to within a single transaction, one database might be committed and the other rolled back in the event of a failure.

There is a compile-time limit of 10 attached database files by default. This can be increased to up to 255 concurrent databases by modifying the following line in `src/sqliteInt.h`:

```
#define MAX_ATTACHED 10
```

To detach an attached database, the syntax is simply

```
DETACH [DATABASE] database-name
```

Manipulating Data

Next we'll look at how records can be added to a database and demonstrate different ways of using the `INSERT` command, and examine the syntax of the SQL `UPDATE` and `DELETE` commands.

Transactions

Any change to a SQLite database must take place within a *transaction*—a block of one or more statements that alter the database in some way. Transactions are the way in which a robust database system ensures that either all or none of the requests to alter the database is carried out; it can never be just partially completed. This property of a database is called *atomicity*.

Whenever an `INSERT`, `UPDATE`, or `DELETE` command is issued, SQLite will begin a new transaction unless one has already been started. An implicit transaction lasts only for the duration of the one statement but ensures that, for instance, an `UPDATE` affecting many rows of a large table will always carry out the action on every row or—in the unlikely event of a system failure while processing this command—none of them. The database will not reflect a change to any row until every row has been updated and the transaction closed.

A transaction can be started from SQL if you want to make a series of changes to the database as one atomic unit. This is the syntax of the `BEGIN TRANSACTION` statement:

```
BEGIN [TRANSACTION [name]] [ON CONFLICT conflict-algorithm]
```

The transaction name is optional and, currently, is ignored by SQLite. The facility to provide a transaction name is included for future use if the ability to nest transactions is added. Currently only one transaction can be open at a time. In fact the keyword TRANSACTION is also optional, but is included for readability.

An ON CONFLICT clause can be specified to override the default conflict resolution algorithm specified at the table level, but can be superseded itself by the OR clause of an INSERT, UPDATE, or DELETE statement.

To end a transaction and save changes to the database, use COMMIT TRANSACTION. The optional transaction name may be specified. To abort a transaction without any of the changes being stored, use ROLLBACK TRANSACTION.

Inserting Data

There are two versions of the syntax for the INSERT statement, depending on where the data to be inserted is coming from.

The first syntax is the one we have already used in Chapter 2, to insert a single row from values provided in the statement itself. The second version is used to insert a dataset returned as the result of a SELECT statement.

INSERT Using VALUES

The syntax for a single-row insert using the VALUES keyword and a list of values provided as part of the statement is as follows:

```
INSERT [OR conflict-algorithm]
INTO [database-name .] table-name [(column-list)]
VALUES (value-list)
```

Although all our examples so far have included a column-list, it is actually optional. Where no column-list is provided, the value-list is assumed to contain one value for each column in the table, in the order they appear in the schema.

This can be a useful shortcut when you are adding data; for instance because we know the column-list is a name and then a color, a record can be inserted into the vegetables table simply using this format:

```
sqlite> INSERT INTO vegetables VALUES ('mushroom', 'white');
```

However if the schema of the table is not what you are expecting, the INSERT will fail with an error. SQLite would not make any assumption as to which columns you are referring to.

Because SQLite does not have an ALTER TABLE command, it is much harder to change a schema after a table has been created than it is with other database engines that include this command. We'll see a workaround for ALTER TABLE in the following example, and if for any reason the vegetables table had been expanded to include three columns, the same INSERT statement would produce this error:

```
sqlite> INSERT INTO vegetables VALUES ('mushroom', 'white');
SQL error: table vegetables has 3 columns but 2 values were supplied
```

Therefore it is good practice to always include the `column-list` in an `INSERT` statement—also known as performing a *full insert.*

The `OR` keyword is used to specify a conflict resolution algorithm in the same way we saw for the `CREATE TABLE` statement. The list of algorithms and their behavior is identical, but the keyword `OR` is used instead of `ON CONFLICT` to give a more natural-sounding syntax.

The conflict algorithm in the `OR` clause of an `INSERT` statement has the highest precedence possible, and will override any other setting present at the table or transaction level.

INSERT Using SELECT

The syntax to insert the result of a `SELECT` query into another table is as follows:

```
INSERT [OR conflict-algorithm]
INTO [database-name .] table-name [(column-list)] select-statement
```

The `select-statement` should return a dataset with the same number and order as the columns specified in the `column-list` (or every column in the destination table if no column-list is supplied). The full syntax of the `SELECT` statement is available, and any number of rows can be returned.

As with `INSERT ... VALUES`, the `column-list` is optional but including it is highly advisable.

Updating Data

The syntax of the `UPDATE` statement in SQLite is as follows:

```
UPDATE [OR conflict-algorithms] [database-name .] table-name
SET assignment [, assignment]*
[WHERE expr]
```

One or more `assignments` can be performed within the same statement upon the same subset of data, defined by the optional `WHERE` clause. An assignment is defined as

```
column-name = expr
```

Although the `WHERE` clause is not required, it is usually desirable. The following example would assign the value of `color` to `green` for every row in the table, when in fact we probably only meant to update one or a few records.

```
sqlite> UPDATE vegetables
   ...> SET color = 'green';
```

The `WHERE` clause can be as simple or complex as necessary, and all the conditional elements that can be used in the `WHERE` clause of a `SELECT` statement can be used here.

It is not logical to join two or more tables when performing an `UPDATE`; however, subselects can be used in the `WHERE` clause, as in the following example:

```
sqlite> UPDATE mytable
   ...> SET myfield = 'somevalue'
   ...> WHERE mykey IN (
   ...>   SELECT keyfield
   ...>   FROM anothertable
   ...> );
```

The OR keyword is used in an UPDATE statement to specify a conflict resolution algorithm with the highest precedence possible, in the same way as with an INSERT.

Deleting Data

The DELETE statement is used to remove rows from a database. Its syntax is

```
DELETE FROM [database-name .] table-name [WHERE expr]
```

As with the UPDATE statement, the WHERE clause is optional but is usually desired—performing a DELETE on a table with no WHERE clause will empty the table. No column-list is required for a DELETE because the operation affects the entire row.

The WHERE clause can use the AND and OR operators to combine conditions and can use subselects to perform a DELETE operation conditional on the results of another query.

The following example modifies a query from Chapter 2 to remove records from the timesheets table where the project_code field does not correspond to a key in the projects table.

```
sqlite> DELETE FROM timesheets
   ...> WHERE project_code NOT IN (
   ...>   SELECT code
   ...>   FROM projects
   ...> );
```

Altering a Table Schema

There is no ALTER TABLE statement in SQLite; instead a table must be dropped and re-created with a new field added, with any data that you want to preserve extracted before the table is dropped and reloaded into the new structure.

A temporary table is the ideal place to hold such data, and the CREATE TABLE ... AS syntax gives us a very easy way to create a copy of an existing table. The syntax is simply

```
CREATE [TEMP | TEMPORARY TABLE] table-name AS select-statement
```

Let's suppose we want to add a new descriptive column to our vegetables table but without losing the data we have already created. The first step is to take a copy of the existing vegetables table to a new temporary table.

```
sqlite> CREATE TEMPORARY TABLE veg_temp
   ...> AS SELECT * FROM vegetables;
```

However, only the field specification has been copied when a table is created this way. The schema of the new table does not include any data type names or column constraints. It is not possible to give a set of column definitions when using `CREATE TABLE ... AS` in SQLite.

```
sqlite> .schema veg_temp
CREATE TEMP TABLE veg_temp(name,color);
```

Compare this to the schema of the original `vegetables` table, which we'll need for re-creating the table with the new field:

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

So now we can safely drop the old `vegetables` table and re-create it with our new field:

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

Finally, reinstate the copied data from the temporary table using the `INSERT ... SELECT` syntax:

```
sqlite> INSERT INTO vegetables (name, color)
   ...> SELECT name, color FROM veg_temp;
```

Loading Data from a File

The `COPY` command in SQLite was based on a similar command found in PostgreSQL and as a result is designed to read the output of the `pg_dump` command to facilitate data transfer between the two systems.

However, `COPY` can also be used to load data from most delimited text file formats into SQLite. It has the following syntax:

```
COPY [OR conflict-algorithm] [database-name .] table-name FROM filename
[USING DELIMITERS delim]
```

The destination table `table-name` must exist—though it need not be empty—before the `COPY` operation is attempted, and either `filename` must be in the current directory or a full path given.

Each line in the input file will become a record in the table, with each column separated by a tab character, unless a different delimiter character is specified in the `USING DELIMITERS` clause.

If a tab—or the specified delimiter—appears within a data column, it must be escaped with a backslash character. The backslash itself can appear in the data if it is escaped itself; in other words it will appear as two consecutive backslash characters.

The special character sequence `\N` in the data file can be used to represent a `NULL` value.

Tab is used to separate columns in the output of `pg_dump`, so it is the default delimiter for the `COPY` command. Another popular format is comma-separated values (CSV). Listing 3.1 shows a comma-separated data file that can be loaded into the three-column `vegetables` table.

Listing 3.1 `vegetables.csv`

```
cucumber,green,Long green salad vegetable
pumpkin,orange,Great for Halloween
avocado,green,Can't make guacamole without it
```

The `COPY` command to load this data file into SQLite is

```
sqlite> COPY vegetables FROM 'vegetables.csv'
   ...> USING DELIMITERS ',';
```

You can instruct `COPY` to read data from the standard input stream instead of a file by using the keyword `STDIN` instead of a filename. A blank line, or a backslash followed by a period, is used to indicate the end of the input.

`COPY` permits an overriding conflict resolution algorithm to be specified after the `OR` keyword, as with `INSERT` and `UPDATE`.

Indexes

The subject of keys and indexes and how they can affect the performance of your database will be addressed in Chapter 4, "Query Optimization," but first we will examine the syntax for creating and finding information on table indexes.

Creating and Dropping Indexes

The `CREATE INDEX` command is used to add a new index to a database table, using this syntax:

```
CREATE [UNIQUE] INDEX index-name
ON [database-name .] table-name (column-name [, column-name]*)
[ON CONFLICT conflict-algorithm]
```

The `index-name` is a user-provided identifier for the new index and must be unique across all database objects. It cannot take the same name as a table, view, or trigger. A popular naming convention is to use the table name and the column name(s) used for the index key separated by an underscore character.

To add an index to the color column of the `vegetables` table, we would use the following command.

```
sqlite> CREATE INDEX vegetables_color
   ...> ON vegetables(color);
```

The syntax of `column-name` allows for a sort order to be given after each column name, either `ASC` or `DESC`; however, currently in SQLite this is ignored. At the present time, all indexes are created in ascending order.

Removing an index is done with reference to the identifier given when it was created, which you can always find by querying the `sqlite_master` table if you cannot remember it.

```
sqlite> SELECT * FROM sqlite_master
   ...> WHERE type = 'index';
    type = index
    name = vegetables_color
tbl_name = vegetables
rootpage = 10
     sql = CREATE INDEX vegetables_color
ON vegetables(color)
```

The `DROP INDEX` command works as you might expect:

```
sqlite> DROP INDEX vegetables_color;
```

Don't worry if you misread the `sqlite_master` output and use the table name instead of the index name. SQLite only allows you to drop indexes with the `DROP INDEX` command and tables with the `DROP TABLE` command.

```
sqlite> DROP INDEX vegetables;
SQL error: no such index: vegetables
```

UNIQUE Indexes

The `UNIQUE` keyword is used to specify that every value in an indexed column is unique. Where an index is created on more than one column, every permutation of the column values has to be unique, even though the same value may appear more than once in its own column.

Since we have already inserted several vegetables of the same color into the table, SQLite will give an error if we attempt to create a unique index on the `color` field.

```
sqlite> CREATE UNIQUE INDEX vegetables_color
   ...> ON vegetables(color);
SQL error: indexed columns are not unique
```

The `ON CONFLICT` clause at the index level is only relevant for a `UNIQUE` index; otherwise, there will never be a conflict on the data it applies to. The conflict resolution algorithm is used when an `INSERT`, `UPDATE`, or `COPY` statement would cause the unique constraint of the index to be violated. It cannot be used in the preceding CREATE UNIQUE INDEX statement to force a unique index onto a column containing multiple values.

The default conflict resolution algorithm is ABORT, and the same list of algorithms is permitted for indexes as in the CREATE TABLE statement.

Views

A view is a convenient way of packaging a query into an object that can itself be used in the FROM clause of a SELECT statement.

Creating and Dropping Views

The syntax for CREATE VIEW is shown next.

```
CREATE [TEMP | TEMPORARY] VIEW view-name AS select-statement
```

The select-statement can be as simple or as complex as necessary; it could return the subset of a single table based on a conditional WHERE clause, or join many tables together to form a single object that can be more easily referenced in SQL.

To drop a view, simply use the DROP VIEW statement with the view-name given when it was created.

A view is not a table. You cannot perform an UPDATE, INSERT, COPY, or DELETE on a view, but if the data in one of the source tables changes, those changes are reflected instantly in the view.

Using Views

The following example shows a view based on the demo database tables employees and employee_rates using a query that returns the current rate of pay for each employee.

```
sqlite> CREATE VIEW current_pay AS
   ...> SELECT e.*, er.rate
   ...> FROM employees e, employee_rates er
   ...> WHERE e.id = er.employee_id
   ...> AND er.end_date IS NULL;
```

We can then query the new view directly, even adding a new condition in the process:

```
sqlite> SELECT * FROM current_pay
   ...> WHERE sex = 'M';
id    first_name  last_name   sex  email                     rate
----  ----------  ----------  ---  ------------------------  ------
101   Alex        Gladstone   M    alex@mycompany.com        30.00
103   Colin       Aynsley     M    colin@mycompany.com       25.00
```

The column names in a view are the column names from the table. Where an expression is used, SQLite will faithfully reproduce the expression as the column heading.

```
sqlite> CREATE VIEW veg_upper AS
   ...> SELECT upper(name), upper(color)
   ...> FROM vegetables;
sqlite> SELECT * FROM veg_upper LIMIT 1;
upper(name)|upper(color)
CARROT|GREEN
```

However, the column in the view cannot actually be called `upper(name)`. As shown in the following example, SQLite will attempt to evaluate the `upper()` function on the nonexistent `name` column.

```
sqlite> SELECT upper(name) from veg_upper;
SQL error: no such column: name
```

Column aliases can be used to give an explicit name to a column so that they can be referenced within a subsequent query.

```
sqlite> CREATE VIEW veg_upper AS
   ...> SELECT upper(name) AS uppername, upper(color) AS uppercolor
   ...> FROM vegetables;
sqlite> SELECT * FROM veg_upper
   ...> WHERE uppercolor = 'ORANGE';
uppername|uppercolor
CARROT|ORANGE
PUMPKIN|ORANGE
```

Note

When a view includes two columns with the same name—whether it is the same column selected twice from one table, or once each from two tables that happen to share a column name—SQLite will modify the column names in the view unless aliases are used. A duplicate column will be suffixed with `_1` the first time it appears, `_2` the second time, and so on.

SQLite does not validate the `select-statement` SQL in `CREATE VIEW`. You will only know if there is an error in the `SELECT` when you come to query the new view. The view's `SELECT` statement is effectively substituted into the query at the point where `view-name` appears, so the errors displayed may not appear to reflect the query you typed.

The following example creates a view with a deliberate error—there is no column entitled `shape` in `vegetables`—and shows that the error is not detected until you query the view.

```
sqlite> CREATE VIEW veg_error AS
   ...> SELECT shape FROM vegetables;
sqlite> SELECT * from veg_error;
SQL error: no such column: shape
```

Triggers

A *trigger* is an event-driven rule on a database, where an operation is initiated when some other transaction (event) takes place. Triggers may be set to fire on any `DELETE`, `INSERT`, or `UPDATE` on a particular table, or on an `UPDATE OF` particular columns within a table.

Creating and Dropping Triggers

The syntax to create a trigger on a table is as follows:

```
CREATE [TEMP | TEMPORARY] TRIGGER trigger-name
[BEFORE | AFTER] database-event ON [database-name .]table-name
trigger-action
```

The `trigger-name` is user-specified and must be unique across all objects in the database—it cannot share the same name as a table, view, or index.

The trigger can be set to fire either `BEFORE` or `AFTER` `database-event`; that is, either to pre-empt the transaction and perform its action just before the `UPDATE`, `INSERT`, or `DELETE` takes place, or to wait until the operation has completed and then immediately carry out the required action.

If the `database-event` is specified as `UPDATE OF column-list`, it will create a trigger that will fire only when particular columns are affected. The trigger will ignore changes that do not affect one of the listed columns.

The `trigger-action` is further defined as

```
[FOR EACH ROW | FOR EACH STATEMENT] [WHEN expression]
BEGIN
 trigger-step; [trigger-step;] *
END
```

At present only `FOR EACH ROW` triggers are supported, so each trigger step—which may be an `INSERT`, `UPDATE`, or `DELETE` statement or `SELECT` with a function expression—is performed once for every affected row in the transaction that causes the trigger to fire. The `WHEN` clause can be used to cause a trigger to fire only for rows for which the `WHEN` clause is true. The `WHEN` clause is formed in the same way as the `WHERE` clause in a `SELECT` statement.

The `WHEN` clause and any `trigger-steps` may reference elements of the affected row, both before and after the trigger action is carried out, as `OLD.column-name` and `NEW.column-name` respectively. For an `UPDATE` action both `OLD` and `NEW` are valid. An `INSERT` event can only provide a reference to the `NEW` value, whereas only `OLD` is valid for a `DELETE` event.

An `ON CONFLICT` clause can be specified in a `trigger-step`; however, any conflict resolution algorithm specified in the statement that causes the trigger to fire will override it.

As you might expect, the syntax to drop a trigger is simply

```
DROP TRIGGER [database-name .] table-name
```

If you forget the name of a trigger, you can query `sqlite_master` using `type = 'trigger'` to find all the triggers on the current database.

Using Triggers

In the last chapter we mentioned that triggers could be used to implement a cascading delete, so that rows from a table that referenced a foreign key would also be deleted if the foreign key were deleted from its own table. The trigger in the following example shows how this could be implemented on the demo database to delete entries from the `timesheets` table if the foreign key `project_code` is deleted from the `projects` table.

```
sqlite> CREATE TRIGGER projcode_cascade
   ...> AFTER DELETE ON projects
   ...> BEGIN
   ...>   DELETE FROM timesheets WHERE project_code = OLD.code;
   ...> END;
```

Similarly, we could create a trigger that maintains data integrity—if the project code changes in the `projects` table, the child records in `timesheets` will be updated to reflect the new foreign key value.

```
sqlite> CREATE TRIGGER projcode_update
   ...> AFTER UPDATE OF code ON projects
   ...> BEGIN
   ...>   UPDATE timesheets
   ...>   SET project_code = NEW.code
   ...>   WHERE project_code = OLD.code;
   ...> END;
```

A quick test verifies that this trigger is working as we want it to:

```
sqlite> UPDATE projects
   ...> SET code = 'NEWCODE'
   ...> WHERE code = 'ABCCONS';
sqlite> SELECT count(*)
   ...> FROM timesheets
   ...> WHERE project_code = 'NEWCODE';
count(*)
----------
3
```

Interrupting a Trigger

Within the `trigger-steps` it is possible to interrupt the command that caused the trigger to fire and execute one of the conflict resolution algorithms available in an `ON CONFLICT` clause. This is done using the `RAISE()` function, which can be invoked using a `SELECT` statement as one of the following:

```
RAISE (ABORT, error-message) |
RAISE (FAIL, error-message) |
RAISE (ROLLBACK, error-message) |
RAISE (IGNORE)
```

Issuing an ABORT, FAIL, or ROLLBACK within a trigger will cause the transaction to exit and take the relevant action, and the error-message parameter is returned to the user.

We could use this behavior to prevent a project code from being deleted from the projects table while rows exist in timesheets that use it as a foreign key, rather than the rather destructive cascading delete.

```
sqlite> CREATE TRIGGER projcode_rollback
   ...> BEFORE DELETE ON projects
   ...> WHEN OLD.code IN (
   ...>   SELECT project_code FROM timesheets
   ...> )
   ...> BEGIN
   ...> SELECT RAISE(ROLLBACK, 'Timesheets exist for that project code');
   ...> END;
```

An attempted DELETE will produce an error:

```
sqlite> DELETE FROM projects WHERE code = 'NEWCODE';
SQL error: Timesheets exist for that project code
```

We can also verify that the DELETE transaction was rolled back.

```
sqlite> SELECT * FROM projects
   ...> WHERE code = 'NEWCODE';
code        client_id   title                start_date  due_date
----------  ----------  -------------------  ----------  ----------
NEWCODE     502         Ongoing consultancy  20030601
```

Using RAISE(IGNORE) causes the current trigger to be abandoned; however, any changes made up to that point will be saved and if the trigger was fired as the result of another trigger, that outer trigger's execution will continue.

Creating a Trigger on a View

The syntax for using triggers on views is slightly different than with tables. This functionality is provided as a way of intercepting INSERT, UPDATE, and DELETE operations on a view, usually to simulate that action by executing the actual steps necessary to make the requested data change appear in the view. Because a view may join two or more tables, a number of steps may be required.

The syntax for creating a trigger on a view is

```
CREATE [TEMP | TEMPORARY] TRIGGER trigger-name
INSTEAD OF database-event ON [database-name .] view-name
trigger-action
```

As before, `database-event` may be `DELETE`, `INSERT`, `UPDATE`, or `UPDATE OF column-list`, and the `trigger-action` is one or more SQL operations contained between the keywords `BEGIN` and `END`.

Working with Dates and Times

In our sample database we have chosen to use integers for columns that store a date value, represented by the format `YYYYMMDD`. This format is fairly readable and, because the most significant part (the year) comes first, allows arithmetic comparisons to be performed. For instance just as February 29th 2004 is earlier than March 1st, `20040229` is a smaller number than `20040301`.

This technique is not without its limitations. First, there is no validation on the values stored. Although February 29th is a valid date in the leap year 2004, it does not exist three years out of four and the value `20050229` is not a real date, yet could still be stored in the integer column or compared to a real date.

In fact even if you used a trigger to make the number eight digits long and also fall within a sensible year range, there are many values that could still be stored that do not represent dates on the calendar. Very strict checking would be required in your application program to ensure such date information was valid.

Similarly, you cannot perform date arithmetic using integer dates. Although `20040101 + 7` gives a date seven days later, `20040330 + 7` would give a number that looks like March 37th.

We have not even looked at a data type to store a time value yet, but the same limitations apply if a numeric field is used. SQLite contains a number of functions that allow you to work with both dates and times stored as character strings, allowing you to manipulate the values in useful ways.

Valid Timestring Formats

SQLite is fairly flexible about the format in which you can specify a date and/or time. The valid time string formats are shown in the following list:

- `YYYY-MM-DD`
- `YYYY-MM-DD HH:MM`
- `YYYY-MM-DD HH:MM:SS`
- `YYYY-MM-DD HH:MM:SS.SSS`
- `HH:MM`
- `HH:MM:SS`
- `HH:MM:SS.SSS`
- `now`
- `DDDD.DDDD`

For the format strings that only specify a time, the date is assumed to be `2000-01-01`. Where no time is specified, midday is used. Simply using the string `now` tells SQLite to use the current date and time.

The format string `DDDD.DDDD` represents a Julian day number—the number of days since noon on November 24, 4714 BC, Greenwich Mean Time. SQLite uses Julian date format internally to manipulate date and time values.

Displaying a Formatted Date and Time

The core date and time function in SQLite is `strftime()`, which has the following prototype:

```
strftime(format, timestring, modifier, modifier, ...)
```

This function is based upon the C function `strftime()` and the format parameter will accept most, although not all, of the same conversion specifiers. The following example shows how a date can be reformatted to `MM/DD/YY` format using `strftime()`.

```
sqlite> SELECT strftime('%m/%d/%Y', '2004-10-31');
10/31/2004
```

Table 3.3 lists the conversions that can be performed by SQLite on a timestring.

Table 3.3 **Date and Time Conversion Specifiers**

String	Meaning
`%d`	Day of month, 01-31
`%f`	Fractional seconds, `SS.SSS`
`%H`	Hour, 00-23
`%j`	Day of year, 001-366
`%J`	Julian day number, `DDDD.DDDD`
`%m`	Month, 00-12
`%M`	Minute, 00-59
`%s`	Seconds since 1970-01-01 (unix epoch)
`%S`	Seconds, 00-59
`%w`	Day of week, 0-6 (0 is Sunday)
`%W`	Week of year, 01-53
`%Y`	Year, `YYYY`
`%%`	`%` symbol

Date and Time Modifiers

Given one or more optional `modifier` arguments, `strftime()` can perform a calculation on the date given in `timestring`.

To add or subtract a period of time, the `days`, `hours`, `minutes`, `seconds`, `months` and `years` modifiers can be used, as shown in these examples:

```
sqlite> SELECT strftime('%Y-%m-%d', '2004-10-31', '+7 days');
2004-11-07

sqlite> SELECT strftime('%H:%M', '22:00', '+12 hours');
10:00

sqlite> SELECT strftime('%Y-%m-%d %H:%M:%S',
                        '2004-01-01 00:00:00', '-1 second', '+1 year');
2004-12-31 23:59:59
```

Note

The modifier keywords can be written as either singular or plural. In the last of the preceding examples, we used `1 second` and `1 year` rather than `1 seconds` and `1 years` for readability. SQLite does not understand English grammar, so either is always acceptable.

In these examples we have used the same output format as the original `timestring` to return the date information in a format that can be recognized by SQLite.You should only format the date differently when you want to display it in your application in a particular way.

To save having to enter the same `format` strings repeatedly when working with dates, SQLite provides four convenience functions that call `strftime()` with predefined formats.

Use `date()` to return a date with the format string `%Y-%m-%d` and `time()` to return a time as `%H:%S`.The function `datetime()` returns the date and time using these two formats combined. Finally `julianday()` uses the `%J` format specifier to return the Julian day number.

The arguments to all four functions are the same as `strftime()` except that the `format` argument is omitted.The following example uses `datetime()` to produce a more concise SQL statement:

```
sqlite> SELECT datetime('2004-01-01 00:00:00', '-1 second', '+1 year');
2004-12-31 23:59:59
```

Other modifiers allow you to adjust a date or time to the nearest significant value. Specifying start of month, start of year, or start of day will decrease the value given in timestring to midnight on the first of the month or year, or on that day respectively.

When executed on any day during 2004, the `start of year` modifier returns `2004-01-01`, as shown in the following example:

```
sqlite> SELECT datetime('now', 'start of year');
2004-01-01 00:00:00
```

Modifiers are applied to `timestring` in the order they appear in the statement, as shown in the following example. Note that had the second statement been executed on the last day of the month, the result would have been different—the start of the following month would have been returned.

```
sqlite> SELECT datetime('now', 'start of month', '+1 day');
2004-07-02 00:00:00

sqlite> SELECT datetime('now', '+1 day', 'start of month');
2004-07-01 00:00:00
```

Any number of modifiers can be combined, giving you considerable power when working with dates and times. For instance, the last day of the current month can be found using three modifiers in succession.

```
sqlite> SELECT date('now', '+1 month', 'start of month', '-1 day');
2004-07-31
```

Handling Different Time Zones

The locale settings of your system will determine which time zone is used when displaying dates and times; however, the underlying system clock will use Coordinated Universal Time (UTC), also known as Greenwich Mean Time (GMT)Greenwich Mean Time (GMT). Your time zone setting will specify a number of hours to be added to or subtracted from the UTC value to arrive at the correct local time.

For instance, to find the local time in New York you have to subtract five hours from UTC, or four hours during daylight savings time. Even in Greenwich, the local time is UTC + 1 hour during the summer months.

To convert between UTC and local time values when formatting a date, use the `utc` or `localtime` modifiers. The following examples were run on a system with the timezone set to Eastern Standard Time (UTC – 5 hours).

```
sqlite> SELECT time('12:00', 'localtime');
2000-01-01 07:00:00

sqlite> SELECT time('12:00', 'utc');
2000-01-01 17:00:00
```

SQL92 Features Not Supported

We finish this chapter on SQLite's implementation of the SQL language by looking at features of the ANSI SQL92 standard that are not currently supported by SQLite.

- Although the `CREATE TABLE` syntax permits an optional `CHECK` clause to be present, the `CHECK` constraint is not enforced.
- The keywords `FOREIGN KEY` are allowable in a `CREATE TABLE` statement; however, this currently has no effect.

- Subqueries must return a static data set, and they may not refer to variables in the outer query—also known as correlated subqueries.
- All triggers are currently `FOR EACH ROW`, even if `FOR EACH STATEMENT` is specified.
- Views are read-only, even when they select only from one table. However, an `INSTEAD OF` trigger can fire on an attempted `INSERT`, `UPDATE`, or `DELETE` to a view and deal with the transaction in the desired manner.
- `INSTEAD OF` triggers are allowed only on views, not on tables.
- Recursive triggers—triggers that trigger themselves—are not supported.
- The `ALTER TABLE` statement is not present; instead a table must be dropped and re-created with the new schema.
- Transactions cannot be nested.
- `count(DISTINCT column-name)` cannot be used. However, this can be achieved by selecting a `count()` from a subselect of the desired table that uses the `DISTINCT` keyword.
- All outer joins must be written as `LEFT OUTER JOIN`. `RIGHT OUTER JOIN` and `FULL OUTER JOIN` are not recognized.
- The `GRANT` and `REVOKE` commands are meaningless in SQLite—the only permissions applicable are those on the database file itself.