Using

SQLite

O'REILLY®

Jay A. Kreibich

Table of Contents

Chapter 4. The SQL Language. Using SQLite, ISBN: 9781449394592

CHAPTER 4

The SQL Language

This chapter provides an overview of the *Structured Query Language*, or *SQL*. Although sometimes pronounced "sequel," the official pronunciation is to name each letter as "ess-cue-ell." The SQL language is the main means of interacting with nearly all modern relational database systems. SQL provides commands to configure the tables, indexes, and other data structures within the database. SQL commands are also used to insert, update, and delete data records, as well as query those records to look up specific data values.

All interaction with a relational database is done through the SQL language. This is true when interactively typing commands or when using the programming API. In all cases, data is stored, modified, and retrieved through SQL commands. Many times, people look through the list of API calls, looking for functions that provide direct program access to the table or index data structures. Functions of this sort do not exist. The API is structured around preparing and issuing SQL commands to the database engine. If you want to query a table or insert a value using the API, you must create and execute the proper SQL command. If you want to do relational database programming, you must know SQL.

Learning SQL

The goal of this chapter is to introduce you to all the major SQL commands and show some of the basic usage patterns. The first time you read through this chapter, don't feel you need to absorb everything at once. Get an idea of what structures the database supports, and how they might be used, but don't feel that you need to memorize the details of every last command.

For people just getting started, the most important commands are `CREATE TABLE`, `INSERT`, and `SELECT`. These will let you create a table, insert some data into the table, and then query the data and display it. Once you get comfortable with those commands, you can start to look at the others in more depth. Feel free to refer back to this chapter, or the command reference in Appendix C. The command reference provides detailed descriptions of each command, including some of the more advanced syntax that isn't covered in this chapter.

Always remember that SQL is a command language. It assumes you know what you're doing. If you're directly entering SQL commands through the `sqlite3` application, the program will not stop and ask for confirmation before processing dangerous or destructive commands. When entering commands by hand, it is always worth pausing and looking back at what you've typed before you hit return.

If you are already reasonably familiar with the SQL language, it should be safe to skim this chapter. Much of the information here is on the SQL language in general, but there is some information about the specific dialect of SQL that SQLite recognizes. Again, Appendix C provides a reference to the specific SQL syntax used by SQLite.

Brief Background

Although the first official SQL specification was published in 1986 by the American National Standards Institute (ANSI), the language traces its roots back to the early 1970s and the pioneering relational database work that was being done at IBM. Current SQL standards are ratified and published by the International Standards Organization (ISO). Although a new standard is published every few years, the last significant set of changes to the core language can be traced to the SQL:1999 standard (also known as "SQL3"). Subsequent standards have mainly been concerned with storing and processing XML-based data. Overall, the evolution of SQL is firmly rooted in the practical aspects of database development, and in many cases new standards only serve to ratify and standardize syntax or features that have been present in commercial database products for some time.

Declarative

The core of SQL is a *declarative language*. In a declarative language, you state what you want the results to be and allow the language processor to figure out how to deliver the desired results. Compare this to *imperative languages*, such as C, Java, Perl, or Python, where each step of a calculation or operation must be explicitly written out, leaving it up to the programmer to lead the program, step by step, to the correct conclusion.

The first SQL standards were specifically designed to make the language approachable and usable by "non-computer people"—at least by the 1980s definition of that term. This is one of the reasons why SQL commands tend to have a somewhat English-like

syntax. Most SQL commands take the form *verb-subject*. For example, `CREATE` (verb) `TABLE` (subject), `DROP INDEX`, `UPDATE` *`table_name`*.

The almost-English, declarative nature of SQL has both advantages and disadvantages. Declarative languages tend to be simpler (especially for nonprogrammers) to understand. Once you get used to the general structure of the commands, the use of English keywords can make the syntax easier to remember. The fixed command structure also makes it much easier for the database engine to optimize queries and return data more efficiently.

The predefined nature of declarative statements can sometimes feel a bit limited, however—especially in a command-based language like SQL, where individual, isolated commands are constructed and issued one at a time. If you require a query that doesn't fit into the processing order defined by the `SELECT` command, you may find yourself having to patch together nested queries or temporary tables. This is especially true when the problem you're trying to solve is inherently nonrelational, and you're forced to jump through extra hoops to account for that.

Despite its oddities and somewhat organic evolution, SQL is a powerful language with a surprisingly broad ability to express complex operations. Once you wrap your head around what makes SQL tick, you can often find moderately simple solutions to even the most off-the-beaten-path problems. It can take some adjustment, however, especially if your primary experience is with imperative languages.

Portability

SQL's biggest flaw is that formal standardization has almost always followed common implementations. Almost every database product (including SQLite) has custom extensions and enhancements to the core language that help differentiate it from other products, or expose features or control systems that are not covered by the core SQL standard. Often these enhancements are related to performance enhancements, and can be difficult to ignore.

While this less-strict approach to language purity has allowed SQL to grow and evolve in very practical ways, it means that "real world" SQL portability is not all that practical. If you strictly limit yourself to standardized SQL syntax, you can achieve a moderate degree of portability, but normally this comes at the cost of lower performance and less data integrity. Generally, applications will write to the specific SQL dialect they're using and not worry about cross-database compatibility. If cross-database compatibility is important to a specific application, the normal approach is to develop a core list of SQL commands required by the application, with minor tweaks for each specific database product.

SQLite makes an effort to follow the SQL standards as much as possible. SQLite will also recognize and correctly parse a number of nonstandard syntax conventions used by other popular databases. This can help with the portability issues.

SQL is not without other issues, but considering its lineage, it is surprisingly well suited for the task at hand. Love it or hate it, it is the relational database language of choice, and it is likely to be with us for a long time.

General Syntax

Before getting into specific commands in SQL, it is worth looking at the general language structure. Like most languages, SQL has a fairly complete expression syntax that can be used to define command parameters. A more detailed description of the expression support can be found in Appendix D.

Basic Syntax

SQL consists of a number of different commands, such as `CREATE TABLE` or `INSERT`. These commands are issued and processed one at a time. Each command implements a different action or feature of the database system.

Although it is customary to use all capital letters for SQL commands and keywords, SQL is a case-insensitive* language. All commands and keywords are case insensitive, as are identifiers (such as table names and column names).

Identifiers must be given as literals. If necessary, identifiers can be enclosed in the standards compliant double-quotes (`" "`) to allow the inclusion of spaces or other non-standard characters in an identifier. SQLite also allows identifiers to be enclosed in square brackets (`[ ]`) or back ticks (`` ` ` ``) for compatibility with other popular database products. SQLite reserves the use of any identifier that uses `sqlite_` as a prefix.

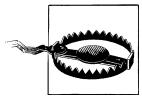
SQL is whitespace insensitive, including line breaks. Individual statements are separated by a semicolon. If you're using an interactive application, such as the `sqlite3` command-line tool, then you'll need to use a semicolon to indicate the end of a statement. The semicolon is not strictly required for single statements, however, as it is properly a statement separator and not a statement terminator. When passing SQL commands into the programming API, the semicolon is not required unless you are passing more than one command statement within a single string.

Single-line comments start with a double dash (`--`) and go to the end of the line. SQL also supports multi-line comments using the C comment syntax (`/* */`).

As with most languages, numeric literals are represented bare. Both integer (`453`) and real (rational) numbers (`43.23`) are recognized, as is exponent-style scientific notation (`9.745e-6`). In order to avoid ambiguities in the parser, SQLite requires that the decimal point is always represented as a period (`.`), regardless of the current internationalization setting.

* Unless otherwise specified, case insensitivity only applies to ASCII characters. That is, characters represented by values less than 128.

Text literals are enclosed in single quotes (`' '`). To represent a string literal that includes a single quote character, use two single quotes in a row (`publisher = 'O''Reilly'`). C-style backslash escapes (`\'`) are not part of the SQL standard and are not supported by SQLite. BLOB literals (binary data) can be represented as an `x` (or `X`) followed by a string literal of hexadecimal characters (`x'A554E59C'`).



Text literals use single quotes. Double quotes are reserved for identifiers (table names, columns, etc.). C-style backslash escapes are not part of the SQL standard.

SQL statements and expressions frequently contain lists. A comma is used as the list separator. SQL does not allow for a trailing comma following the last item of a list.

In general, expressions can be used any place a literal data value is allowed. Expressions can include both mathematical statements, as well as functions. Function-calling syntax is similar to most other computer languages, utilizing the name of the function, followed by a list of parameters enclosed in parentheses. Expressions can be grouped into subexpressions using parentheses.

If an expression is evaluated in the context of a row (such as a filtering expression), the value of a row element can be extracted by naming the column. You may have to qualify the column name with a table name or alias. If you're using cross-database queries, you may also have to specify which database you're referring to. The syntax is:

```
[[database_name.]table_name.]column_name
```

If no database name is given, it is assumed you're referring to the `main` database on the default connection. If the table name/alias is also omitted, the system will make a best-guess using just the column name, but will return an error if the name is ambiguous.

Three-Valued Logic

SQL allows any value to be assigned a *NULL*. NULL is not a value in itself (SQLite actually implements it as a unique valueless type), but is used as a marker or flag to represent unknown or missing data. The thought is that there are times when values for a specific row element may not be available or may not be applicable.

A NULL may not be a value, but it can be assigned to data elements that normally have values, and can therefore show up in expressions. The problem is that NULLs don't interact well with other values. If a NULL represents an unknown that might be any possible value, how can we know if the expression `NULL > 3` is true or false?

To deal with this problem, SQL must employ a concept called *three-valued logic*. Three-valued logic is often abbreviated *TVL* or *3VL*, and is more formally known as *ternary logic*. 3VL essentially adds an "unknown" state to the familiar true/false Boolean logic system.

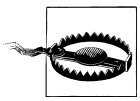
Here are the truth tables for the 3VL operators `NOT`, `AND`, and `OR`:

Value	NOT Value
True	False
False	True
NULL	NULL

3VL AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

3VL OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

3VL also dictates how comparisons work. For example, any equality check (=) involving a NULL will evaluate to NULL, including `NULL = NULL`. Remember that NULL is not a value, it is a flag for the unknown, so the expression `NULL = NULL` is asking, "Does this unknown equal that unknown?" The only practical answer is, "That is unknown." It might, but it might not. Similar rules apply to greater-than, less-than, and other comparisons.



You cannot use the equality operator (=) to test for NULLs. You must use the `IS NULL` operator.

If you're having trouble resolving an expression, just remember that a NULL marks an unknown or unresolved value. This is why the expression `False AND NULL` is false, but `True AND NULL` is NULL. In the case of the first expression, the NULL can be replaced by either true or false without altering the expression result. That isn't true of the second expression, where the outcome is unknown (in other words, NULL) because the output depends on the unknown input.

Simple Operators

SQLite supports the following unary prefix operators:

`- +`
: These adjust the sign of a value. The "-" operator flips the sign of the value, effectively multiplying it by `-1.0`. The "`+`" operator is essentially a no-op, leaving a value with the same sign it previously had. It does not make negative values positive.

`~`
: As in the C language, the "~" operator performs a bit-wise inversion. This operator is not part of the SQL standard.

`NOT`
: The NOT operator reverses a Boolean expression using 3VL.

There are also a number of binary operators. They are listed here in descending precedence.

`||`
: String concatenation. This is the only string concatenation operator recognized by the SQL standard. Many other database products allow "`+`" to be used for concatenation, but SQLite does not.

`+ - * / %`
: Standard arithmetic operators for addition, subtraction, multiplication, division, and modulus (remainder).

`| & << >>`
: The bitwise operators *or*, *and*, and shift-high/shift-low, as found in the C language. These operators are not part of the SQL standard.

`< <= => >`
: Comparison test operators. Again, just as in the C language we have less-than, less-than or equal, greater-than or equal, and greater than. These operators are subject to SQL's 3VL regarding NULLs.

`= == != <>`
: Equality test operators. Both "`=`" and "`==`" will test for equality, while both "`!=`" and "`<>`" test for inequality. Being logic operators, these tests are subject to SQL's 3VL regarding NULLs. Specifically, `value = NULL` will always return NULL.

`IN LIKE GLOB MATCH REGEXP`
: These five keywords are logic operators, returning, true, false, or NULL state. See Appendix D for more specifics on these operators.

`AND OR`
: Logical operators. Again, they are subject to SQL's 3VL.

In addition to these basics, SQL supports a number of specific expression operations. For more information on these and any SQL-specific expressions, see Appendix D.

SQL Data Languages

SQL commands are divided into four major categories, or *languages*. Each language defines a subset of commands that serve a related purpose. The first language is the *Data Definition Language*, or *DDL*, which refers to commands that define the structure of tables, views, indexes, and other data containers and objects within the database. `CREATE TABLE` (used to define a new table) and `DROP VIEW` (used to delete a view) are examples of DDL commands.

The second category of commands is known as *Data Manipulation Language*, or *DML*. These are all of the commands that insert, update, delete, and query actual data values from the data structures defined by the DDL. `INSERT` (used to insert new values into a table) and `SELECT` (used to query or look up data from tables) are examples of DML commands.

Related to the DML and DDL is the *Transaction Control Language*, or *TCL*. TCL commands can be used to control transactions of DML and DDL commands. `BEGIN` (used to begin a multistatement transaction) and `COMMIT` (used to end and accept a transaction) are examples of TCL commands.

The last category is the *Data Control Language*, or *DCL*. The main purpose of the DCL is to grant or revoke access control. Much like file permissions, DCL commands are used to allow (or deny) specific database users (or groups of users) permission to utilize or access specific resources within a database. These permissions can apply to both the DDL and the DML. DDL permissions might include the ability to create a real or temporary table, while DML permissions might include the ability to read, update, or delete the records of a specific table. `GRANT` (used to assign a permission) and `REVOKE` (used to delete an existing permission) are the primary DCL commands.

SQLite supports the majority of standardized DDL, DML, and TCL commands but lacks any DCL commands. Because SQLite does not have user names or logins, it does not have any concept of assigned permissions. Rather, SQLite depends on datatype permissions to define who can open and access a database.

Data Definition Language

The DDL is used to define the structure of data containers and objects within the database. The most common of these containers and objects are tables, indexes, and views. As you'll see, most objects are defined with a variation of the `CREATE` command, such as `CREATE TABLE` or `CREATE VIEW`. The `DROP` command is used to delete an existing object (and all of the data it might contain). Examples include `DROP TABLE` or `DROP INDEX`. Because the command syntax is so different, statements like `CREATE TABLE` or `CREATE INDEX` are usually considered to be separate commands, and not variations of a single `CREATE` command.

In a sense, the DDL commands are similar to C/C++ header files. DDL commands are used to define the structure, names, and types of the data containers within a database, just as a header file typically defines type definitions, structures, classes, and other data structures. DDL commands are typically used to set up and configure a brand new database before data is entered.

DDL commands define the basic structure of the database and are typically run when a new database is created.

DDL commands are often held in a script file, so that the structure of the database can be easily recreated. Sometimes, especially during development, you may need to recreate only part of the database. To help support this, most `CREATE` commands in SQLite have an optional `IF NOT EXISTS` clause.

Normally, a `CREATE` statement will return an error if an object with the requested name already exists. If the optional `IF NOT EXISTS` clause is present, then this error is suppressed and nothing is done, even if the structure of the existing object and the new object are not compatible. Similarly, most `DROP` statements allow an optional `IF EXISTS` clause that silently ignores any request to delete an object that isn't there.

In the examples that follow, the `IF EXISTS` and `IF NOT EXISTS` command variations are not explicitly called out. Please see the SQLite command reference in Appendix C for the full details on the complete syntax supported by SQLite.

Tables

The most common DDL command is `CREATE TABLE`. No data values can be stored in a database until a table is defined to hold that data. At the bare minimum, the `CREATE TABLE` command defines the table name, plus the name of each column. Most of the time you'll want to define a type for each column as well, although types are optional when using SQLite. Optional constraints, conditions, and default values can also be assigned to each column. Table-level constraints can also be assigned, if they involve multiple columns.

In some large RDBMS systems, `CREATE TABLE` can be quite complex, defining all kinds of storage options and configurations. SQLite's version of `CREATE TABLE` is somewhat simpler, but there are still a great many options available. For full explanation of the command, see `CREATE TABLE` in Appendix C.

The basics

The most basic syntax for `CREATE TABLE` looks something like this:

```
CREATE TABLE table_name
(
   column_name  column_type,
   [...]
);
```

A table name must be provided to identify the new table. After that, there is just a simple list of column names and their types. Table names come from a global namespace of all identifiers, including the names of tables, views, and indexes.

Clear and concise identifier names are important. Like the database design itself, some careful thought should be put into the table name, trying to pin down the precise meaning of the data it contains. Much the same could be said of column names. Table names and column names tend to be referenced frequently in queries, so there is some desire to keep them as brief as possible while still keeping their purpose clear.

Column types

Most databases use strong, static column typing. This means that the elements of a column can only hold values compatible with the column's defined type. SQLite utilizes a dynamic typing technique known as *manifest typing*. For each row value, manifest typing records the value's type along with the value data. This allows nearly any element of any row to hold almost any type of value.

In the strictest sense, SQLite supports only five concrete datatypes. These are known as *storage classes*, and represent the different ways SQLite might choose to store data on disk. Every value has one of these five native storage classes:

NULL
: A NULL is considered its own distinct type. A NULL type does not hold a value. Literal NULLs are represented by the keyword `NULL`.

Integer
: A signed integer number. Integer values are variable length, being 1, 2, 3, 4, 6, or 8 bytes in length, depending on the minimum size required to hold the specific value. Integer have a range of –9,223,372,036,854,775,808 to +9,223,372,036,854,775,807, or roughly 19 digits. Literal integers are represented by any bare series of numeric digits (without commas) that does not include a decimal point or exponent.

Float
: A floating-point number, stored as an 8-byte value in the processor's native format. For nearly all modern processors, this is an IEEE 754 double-precision number. Literal floating-point numbers are represented by any bare series of numeric digits that include a decimal point or exponent.

Text
: A variable-length string, stored using the database encoding (UTF-8, UTF-16BE, or UTF-16LE). Literal text values are represented using character strings in single quotes.

BLOB
: A length of raw bytes, copied exactly as provided. Literal BLOBs are represented as hexadecimal text strings preceded by an `x`. For example, the notation `x'1234ABCD'` represents a 4-byte BLOB. BLOB stands for Binary Large OBject.

SQLite text and BLOB values are always variable length. The maximum size of a text or BLOB value is limited by a compile-time directive. The default limit is exactly one billion bytes, or slightly less than a full gigabyte. The maximum value for this directive is two gigabytes.

Since the elements of most columns can hold any value type, the "type" of a column is a somewhat misleading concept. Rather than being an absolute type, as in most databases, an SQLite column type (as defined in `CREATE TABLE`) becomes more of a suggestion than a hard and fast rule. This is known as a *type affinity*, and essentially represents a desired category of type. Each type affinity has specific rules about what types of values it can store, and how different values will be converted when stored in that column. Generally, a type affinity will cause a conversion or migration of types only if it can be done without losing data or precision.

Each table column must have one of five type affinities:

Text
: A column with a text affinity will only store values of type NULL, text, or BLOB. If you attempt to store a value with a numeric type (float or integer) it will be converted into a text representation before being stored as a text value type.

Numeric
: A column with a numeric affinity will store any of the five types. Values with integer and float types, along with NULL and BLOB types, are stored without conversion. Any time a value with a text type is stored, an attempt is made to convert the value to a numeric type (integer or float). Assuming the conversion works, the value is stored in an appropriate numeric type. If the conversion fails, the text value is stored without any type of conversion.

Integer
: A column with an integer affinity works essentially the same as a numeric affinity. The only difference is that any value with a float type that lacks a fractional component will be converted into an integer type.

Float
: A column with a floating-point affinity also works essentially the same as a numeric affinity. The only difference is that most values with integer types are converted into floating-point values and stored as a float type.

None
: A column with a none affinity has no preference over storage class. Each value is stored as the type provided, with no attempt to convert anything.

Since type affinities are not part of the SQL standard, SQLite has a series of rules that attempt to map traditional column types to the most logical type affinity. The type affinity of a column is determined by the declared type of the column, according to the following rules (substring matches are case-insensitive):

1. If no column type was given, then the column is given the none affinity.
2. If the column type contains the substring "INT," then the column is given the integer affinity.
3. If the column type contains any of the substrings "CHAR," "CLOB," or "TEXT," then the column is given the text affinity.
4. If the column type contains the substring "BLOB," then the column is given the none affinity.
5. If the column type contains any of the substrings "REAL," "FLOA," or "DOUB," then it is given the float affinity.
6. If no match is found, the column is assigned the numeric affinity.

As implied by the first rule, the column type is completely optional. SQLite will allow you to create a table by simply naming the columns, such as `CREATE TABLE t ( i, j, k);`. You'll also notice that there isn't any specific list of column types that are recognized. You can use any column type you want, even making up your own names.

This might sound a bit fast and loose for a typing system, but it works out quite well. By keying off specific substrings, rather than trying to define specific types, SQLite is able to handle SQL statements (and their database-specific types) from just about any database, all while doing a pretty good job of mapping the types to an appropriate affinity. About the only type you need to be careful of is "floating point." The "int" in "point" will trigger rule 2 before the "floa" in "floating" will get to rule 5, and the column affinity will end up being *integer*.

Column constraints

In addition to column names and types, a table definition can also impose constraints on specific columns or sets of columns. A more complete view of the `CREATE TABLE` command looks something like this:

```
CREATE TABLE table_name
(
   column_name  column_type    column_constraints...,
   [... ,]

   table_constraints,
   [...]
);
```

Here we see that each individual column can have additional, optional constraints and modifiers placed on it. Column constraints only affect the column for which they are defined, while table constraints can be used to define a constraint across one or more columns. Constraints that involve two or more columns must be defined as table constraints.

Column constraints can be used to define a custom sort order (`COLLATE` *`collation_name`*). Collations determine how text values are sorted. In addition to user-defined collations, SQLite includes a `NOCASE` collation that ignores case when sorting.

A default value (`DEFAULT` *`value`*) can also be assigned. Nearly all columns have a default of NULL. You can use the `DEFAULT` column constraint to set a different default value. The default can either be a literal or an expression. Expressions must be placed in parentheses.

To help with date and time defaults, SQLite also includes three special keywords that may be used as a default value: `CURRENT_TIME`, `CURRENT_DATE`, and `CURRENT_TIMESTAMP`. These will record the UTC time, date, or date and time, respectively, when a new row is inserted. See "Date and Time Features" on page 159 for more information on date and time functions.

Column constraints can also impose limits on a column, like denying NULL (`NOT NULL`) or requiring a unique value for each row (`UNIQUE`). Remember that a NULL is not considered a value, so `UNIQUE` does not imply `NOT NULL`, nor does `UNIQUE` imply only a single NULL is allowed. If you want to keep NULL assignments out of a `UNIQUE` column, you need to explicitly mark the column `NOT NULL`.

Column values can also be subjected to arbitrary user-defined constraints before they are assigned (`CHECK ( expression )`). Some types of constraints also allow you to specify the action to be taken in situations when the constraint would be violated. See `CREATE TABLE` and `UPDATE` in Appendix C for more specific details.

When multiple column constraints are used on a single column, the constraints are listed one after another without commas. Some examples:

```
CREATE TABLE parts
(
    part_id    INTEGER   PRIMARY KEY,
    stock      INTEGER   DEFAULT 0   NOT NULL,
    desc       TEXT      CHECK( desc != '' )    -- empty strings not allowed
);
```

In order to enforce a `UNIQUE` column constraint, a unique index will be automatically created over that column. A different index will be created for each column (or set of columns) marked `UNIQUE`. There is some expense in maintaining an index, so be aware that enforcing a `UNIQUE` column constraint can have performance considerations.

Primary keys

In addition to these other constraints, a single column (or set of columns) can be designated as the `PRIMARY KEY`. Each table can have only one primary key. Primary keys must be unique, so designating a column as `PRIMARY KEY` implies the `UNIQUE` constraint as well, and will result in an automatic unique index being created. If a column is marked both `UNIQUE` and `PRIMARY KEY`, only one index will be created.

In SQLite, `PRIMARY KEY` does not imply `NOT NULL`. This is in contradiction to the SQL standard and is considered a bug, but the behavior is so long-standing that there are concerns about fixing it and breaking existing applications. As a result, it is always a good idea to explicitly mark at least one column from each `PRIMARY KEY` as `NOT NULL`.

There are also some good design reasons for defining a primary key, which will be discussed in "Tables and Keys" on page 87, but the only significant, concrete thing that comes out of defining a `PRIMARY KEY` is the automatic unique index. There are also some minor syntax shortcuts.

If, however, the primary key column has a type that is designated as `INTEGER` (and very specifically `INTEGER`), then that column becomes the table's "root" column.

SQLite must have some column that can be used to index the base storage for the table. In a sense, that column acts as the master index that is used to store the table itself. Like many other database systems, SQLite will silently create a hidden `ROWID` column for this purpose. Different database systems use different names so, in an effort to maintain compatibility, SQLite will recognize the names `ROWID`, `OID`, or `_ROWID_` to reference the root column. Normally `ROWID` columns are not returned (even for column wildcards), nor are their values included in dump files.

If a table includes an `INTEGER PRIMARY KEY` column, then that column becomes an alias for the automatic `ROWID` column. You can still reference the column by any of the `ROWID` names, but you can also reference the column by its "real" user-defined name. Unlike `PRIMARY KEY` by itself, `INTEGER PRIMARY KEY` columns do have an automatic `NOT NULL` constraint associated with them. They are also strictly typed to only accept integer values.

There are two significant advantages of `INTEGER PRIMARY KEY` columns. First, because the column aliases the table's root `ROWID` column, there is no need for a secondary index. The table itself acts as the index, providing efficient lookups without the maintenance costs of an external index.

Second, `INTEGER PRIMARY KEY` columns can automatically provide unique default values. When you insert a row without an explicit value for the `ROWID` (or `ROWID` alias) column, SQLite will automatically choose a value that is one greater than the largest existing value in the column. This provides an easy means to automatically generate unique keys. If the maximum value is reached, the database will randomly try other values, looking for an unused key.

`INTEGER PRIMARY KEY` columns can optionally be marked as `AUTOINCREMENT`. In that case, the automatically generated ID values will constantly increase, preventing the reuse of an ID value from a previously deleted row. If the maximum value is reached, insertions with automatic `INTEGER PRIMARY KEY` values are no longer possible. This is unlikely, however, as the `INTEGER PRIMARY KEY` type domain is large enough to allow 1,000 inserts per second for almost 300 million years.

When using either automatic or `AUTOINCREMENT` values, it is always possible to insert an explicit `ROWID` (or `ROWID` alias) value. Other than the `INTEGER PRIMARY KEY` designation, SQLite offers no other type of automatic sequence feature.

In addition to a `PRIMARY KEY`, columns can also be marked as a `FOREIGN KEY`. These columns reference rows in another (foreign) table. Foreign keys can be used to create links between rows in different tables. See "Tables and Keys" on page 87 for details.

Table constraints

Table definitions can also include table-level constraints. In general, table constraints and column constraints work the same way. Table-level constraints still operate on individual rows. The main difference is that using the table constraint syntax, you can apply the constraint to a group of columns rather than just a single column. It is perfectly legal to define a table constraint with only one column, effectively defining a column constraint. Multicolumn constraints are sometimes known as *compound constraints*.

At the table level, SQLite supports the `UNIQUE`, `CHECK`, and `PRIMARY KEY` constraints. The check constraint is very similar, requiring only an expression (`CHECK (`*`expression`*`)`). Both the `UNIQUE` and `PRIMARY KEY` constraints, when given as a table constraint, require a list of columns (e.g., `UNIQUE (`*`column_name`*`, [...])`, `PRIMARY KEY (`*`column_name`*`, [...])`). As with column constraints, any table-level `UNIQUE` or `PRIMARY KEY` (which implies `UNIQUE`) constraint will automatically create a unique index over the appropriate columns.

Table constraints that are applied to multiple columns use the set of columns as a group. For example, when `UNIQUE` or `PRIMARY KEY` is applied across more than one column, each individual column is allowed to have duplicate values. The constraint only prevents the set of values across the designated columns from being replicated. If you wanted each individual column to also be `UNIQUE`, you'd need to add the appropriate constraints to the individual columns.

Consider a table that contains records of all the rooms in a multibuilding campus:

```
CREATE TABLE rooms
(
    room_number       INTEGER  NOT NULL,
    building_number   INTEGER  NOT NULL,
    [...,]

    PRIMARY KEY( room_number, building_number )
);
```

Clearly we need to allow for more than one room with the number 101. We also need to allow for more than one room in building 103. But there should only be one room 101 in building 103, so we apply the constraint across both columns. In this example, we've chosen to make these columns into a compound primary key, since the building number and room number combine to quintessentially define a specific room. Depending on the design of the rest of the database, it might have been equally valid to define a simple `UNIQUE` constraint across these two columns, and designated an arbitrary `room_id` column as the primary key.

Tables from queries

You can also create a table from the output of a query. This is a slightly different `CREATE TABLE` syntax that creates a new table and preloads it with data, all in one command:

```
CREATE [TEMP] TABLE table_name AS SELECT query_statement;
```

Using this form, you do not designate the number of columns or their names or types. Rather, the query statement is run and the output is used to define the column names and preload the new table with data. With this syntax, there is no way to designate column constraints or modifiers. Any result column that is a direct column reference will inherit the column's affinity, but and all columns are given a `NONE` affinity. The query statement consists of a `SELECT` command. More information on `SELECT` can be found in Chapter 5.

Tables created in this manner are not dynamically updated—the query command is run only once when the table is created. Once the data is entered into the new table, it remains unaltered until you change it. If you need a table-like object that can dynamically update itself, use a `VIEW` ("Views" on page 43).

This example shows the optional `TEMP` keyword (the full word `TEMPORARY` can also be used) in `CREATE TEMP TABLE`. This modifier can be used on any variation of `CREATE TABLE`, but is frequently used in conjunction with the `...AS SELECT...` variation shown here. Temporary tables have two specific features. First, temporary tables can only be seen by the database connection that created them. This allows the simultaneous reuse of table names without any worry of conflict between different clients. Second, all associated temporary tables are automatically dropped and deleted whenever a database connection is closed.

Generally speaking, `CREATE TABLE...AS SELECT` is not the best choice for creating standard tables. If you need to copy data from an old table into a new table, a better choice is to use `CREATE TABLE` to define an empty table with all of the appropriate column modifiers and table constraints. You can then bulk copy all the data into the new table using a variation of the `INSERT` command that allows for query statements. See "INSERT" on page 46 for details.

Altering tables

SQLite supports a limited version of the `ALTER TABLE` command. Currently, there are only two operations supported by `ALTER TABLE`: *add column* and *rename*. The add column variant allows you to add new columns to an existing table. It cannot remove them. New columns are always added to the end of the column list. Several other restrictions apply.

If you need to make a more significant change while preserving as much data as possible, you can use the rename variant to rename the existing table, create a new table under the original name, and then copy the data from the old table to the new table. The old table can then be safely dropped.

For full details, see `ALTER TABLE` in Appendix C.

Dropping tables

The `CREATE TABLE` command is used to create tables and `DROP TABLE` is used to delete them. The `DROP TABLE` command deletes a table and all of the data it contains. The table definition is also removed from the database system catalogs.

The `DROP TABLE` command is very simple. The only argument is the name of the table you wish to drop:

```
DROP TABLE table_name;
```

In addition to deleting the table, `DROP TABLE` will also drop any indexes associated with the table. Both automatically created indexes (such as those used to enforce a `UNIQUE` constraint) as well as manually created indexes will be dropped.

Virtual tables

Virtual tables can be used to connect any data source to SQLite, including other databases. A virtual table is created with the `CREATE VIRTUAL TABLE` command. Although very similar to `CREATE TABLE`, there are important differences. For example, virtual tables cannot be made temporary, nor do they allow for an `IF NOT EXISTS` clause. To drop a virtual table, you use the normal `DROP TABLE` command.

For more information on virtual tables, including the full syntax for `CREATE VIRTUAL TABLE`, see Chapter 10.

Views

Views provide a way to package queries into a predefined object. Once created, views act more or less like read-only tables. Just like tables, new views can be marked as `TEMP`, with the same result. The basic syntax of the `CREATE VIEW` command is:

```
CREATE [TEMP] VIEW view_name AS SELECT query_statement
```

The `CREATE VIEW` syntax is almost identical to the `CREATE TABLE...AS SELECT` command. This is because both commands serve a similar purpose, with one important difference. The result of a `CREATE TABLE` command is a new table that contains a full copy of the data. The `SELECT` statement is run exactly once and the output of the query is stored in the newly defined table. Once created, the table will hold its own, independent copy of the data.

A view, on the other hand, is fully dynamic. Every time the view is referenced or queried, the underlying `SELECT` statement is run to regenerate the view. This means the data seen in a view automatically updates as the data changes in the underlying tables. In a sense, views are almost like named queries.

Views are commonly used in one of two ways. First, they can be used to package up commonly used queries into a more convenient form. This is especially true if the query is complex and prone to error. By creating a view, you can be sure to get the same query each time.

Views are also commonly used to create user-friendly versions of standard tables. A common example are tables with date and time records. Normally, any time or date value is recorded in Coordinated Universal Time, or UTC. UTC is a more proper format for dates and times because it is unambiguous and time-zone independent. Unfortunately, it can also be a bit confusing if you're several time zones away. It is often useful to create a view that mimics the base table, but converts all the times and dates from UTC into the local time zone. This way the data in the original tables remains unchanged, but the presentation is in units that are more user-friendly.

Views are dropped with the `DROP VIEW` command:

```
DROP VIEW view_name;
```

Dropping a view will not have any effect on the tables it references.

Indexes

Indexes (or indices) are a means to optimize database lookups by pre-sorting and indexing one or more columns of a table. Ideally, this allows specific rows in a table to be found without having to scan every row in the table. In this fashion, indexes can provide a large performance boost to some types of queries. Indexes are not free, however, requiring updates with each modification to a table as well as additional storage space. There are even some situations when an index will cause a drop in performance. See "Indexes" on page 107 for more information on when it makes sense to use an index.

The basic syntax for creating an index specifies the name of the new index, as well as the table and column names that are indexed. Indexes are always associated with one (and only one) table, but they can include one or more columns from that table:

```
CREATE [UNIQUE] INDEX index_name ON table_name ( column_name [, ...] );
```

Normally indexes allow duplicate values. The optional `UNIQUE` keyword indicates that duplicate entries are not allowed, and any attempt to insert or update a table with a nonunique value will cause an error. For unique indexes that reference more than one column, all the columns must match for an entry to be considered duplicate. As discussed with `CREATE TABLE`, NULL isn't considered a value, so a `UNIQUE` index will not prevent one or more NULLs. If you want to prevent NULLs, you must indicate `NOT NULL` in the original table definition.

Each index is tied to a specific table, but they all share a common namespace. Although you can name an index anything you like, it is standard practice to name an index with a standard prefix (such as `idx_`), and then include the table name and possibly the names of the columns included in the index. For example:

```
CREATE INDEX idx_employees_name ON employees ( name );
```

This makes for long index names but, unlike table or view names, you typically only reference an index's name when you create it and when you drop it.

As with all the `DROP` commands, the `DROP INDEX` command is very simple, and requires only the name of the index that is being dropped:

```
DROP INDEX index_name;
```

Dropping an index will remove the index from the database, but will leave the associated table intact.

As described earlier, the `CREATE TABLE` command will automatically create unique indexes to enforce a `UNIQUE` or `PRIMARY KEY` constraint. All automatic indexes will start with an `sqlite_` prefix. Because these indexes are required to enforce the table definition, they cannot be manually dropped with the `DROP INDEX` command. Dropping the automatic indexes would alter the table behavior as defined by the original `CREATE TABLE` command.

Conversely, if you have manually defined a `UNIQUE` index, dropping that index will allow the database to insert or update redundant data. Be careful when auditing indexes and remember that not all indexes are created for performance reasons.

Data Manipulation Language

The Data Manipulation Language is all about getting user data in and out of the database. After all the data structures and other database objects have been created with DDL commands, DML commands can be used to load those data structures full of useful data.

The DML supported by SQLite falls into two basic categories. The first category consists of the "update" commands, which includes the actual `UPDATE` command, as well as the `INSERT` and `DELETE` commands. As you might guess, these commands are used to update (or modify), insert, and delete the rows of a table. All of these commands alter

the stored data in some way. The update commands are the primary means of managing all the data within a database.

The second category consists of the "query" commands, which are used to extract data from the database. Actually, there is only one query command: `SELECT`. The `SELECT` command not only prints returned values, but provides a great number of options to combine different tables and rows and otherwise manipulate data before returning the final result.

`SELECT` is, unquestionably, the most complex SQL command. It is also, arguably, the most important SQL command. This chapter will only cover the very basics of `SELECT`, and then we will spend the next chapter going through all of its parts, bit by bit. To address the full command syntax in detail, `SELECT` gets a whole chapter to itself (Chapter 5).

Row Modification Commands

There are three commands used for adding, modifying, and removing data from the database. `INSERT` adds new rows, `UPDATE` modifies existing rows, and `DELETE` removes rows. These three commands are used to maintain all of the actual data values within the database. All three update commands operate at a row level, adding, altering, or removing the specified rows. Although all three commands are capable of acting on multiple rows, each command can only directly act upon rows contained within a single table.

INSERT

The `INSERT` command is used to create new rows in the specified table. There are two meaningful versions of the command. The first version uses a `VALUES` clause to specify a list of values to insert:

```
INSERT INTO table_name (column_name [, ...]) VALUES (new_value [, ...]);
```

A table name is provided, along with a list of columns and a list of values. Both lists must have the same number of items. A single new row is created and each value is recorded into its respective column. The columns can be listed in any order, just as long as the list of columns and the list of values line up correctly. Any columns that are not listed will receive their default values:

```
INSERT INTO parts ( name, stock, status ) VALUES ( 'Widget', 17, 'IN STOCK' );
```

In this example, we attempt to insert a new row into a "parts" table. Note the use of single quotes for text literals.

Technically, the list of column names is optional. If no explicit list of columns is provided, the `INSERT` command will attempt to pair up values with the table's full list of columns:

```
INSERT INTO table_name VALUES (new_value [, ...]);
```

The trick with this format is that the number and order of values must exactly match the number and order of columns in the table definition. That means it is impossible to use default values, even on `INTEGER PRIMARY KEY` columns. More often than not, this is not actually desirable. This format is also harder to maintain within application source code, since it must be meticulously updated if the table format changes. In general, it is recommended that you always explicitly list out the columns in an `INSERT` statement.

When bulk importing data, it is common to loop over data sets, calling `INSERT` over and over. Processing these statements one at a time can be fairly slow, since each command will update both the table and any relevant indexes, and then make sure the data is fully written out to physical disk before (finally!) starting the next `INSERT`. This is a fairly lengthly process, since it requires physical I/O.

To speed up bulk inserts, it is common to wrap groups of 1,000 to 10,000 `INSERT` statements into a single transaction. Grouping the statement together will substantially increase the overall speed of the inserts by delaying the physical I/O until the end of the transaction. See "Transaction Control Language" on page 51 for more information on transactions.

Bulk inserts can be sped up by wrapping large groups of `INSERT` commands inside a transaction.

The second version of `INSERT` allows you to define values by using a query statement. This is very similar to the `CREATE TABLE...AS SELECT` command, although the table must already exist. This is the only version of `INSERT` that can insert more than one row with a single command:

```
INSERT INTO table_name (column_name, [...]) SELECT query_statement;
```

This type of `INSERT` is most commonly used to bulk copy data from one table to another. This is a common operation when you need to update the definition of a table, but you don't want to lose all the data that already exists in the database. The old table is renamed, the new table is defined, and the data is copied from the old table into the new table using an `INSERT INTO...SELECT` command. This form can also be used to populate temporary tables or copy data from one attached database to another.

As with the `VALUES` version of `INSERT`, the column list is technically optional but, for all the same reasons, it is still recommended that you provide an explicit column list.

All versions of the `INSERT` command also support an optional conflict resolution clause. This conflict clause determines what should be done if the results of the `INSERT` would violate a database constraint. The most common example is `INSERT OR REPLACE`, which comes into play when the `INSERT` would, as executed, cause a `UNIQUE` constraint violation. If the `REPLACE` conflict resolution is present, any existing row that would cause a

UNIQUE constraint violation is first deleted, and then the INSERT is allowed to continue. This specific usage pattern is so common that the whole INSERT OR REPLACE phrase can be replaced by just REPLACE. For example, REPLACE INTO *table_name*....

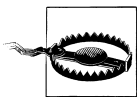
See INSERT and UPDATE in Appendix C for more information on the details of conflict resolution.

UPDATE

The UPDATE command is used to assign new values to one or more columns of existing rows in a table. The command can update more than one row, but all of the rows must be part of the same table. The basic syntax is:

```
UPDATE table_name SET column_name=new_value [, ...] WHERE expression
```

The command requires a table name followed by a list of column name/value pairs that should be assigned. Which rows are updated is determined by a conditional expression that is tested against each row of the table. The most common usage pattern uses the expression to check for equality on some unique column, such as a PRIMARY KEY column.



If no WHERE condition is given, the UPDATE command will attempt to update the designated columns in *every* row of a table.

It is not considered an error if the WHERE expression evaluates to false for every row in the table, resulting in no actual updates.

Here is a more specific example:

```
-- Update the price and stock of part_id 454:
UPDATE parts SET price = 4.25, stock = 75 WHERE part_id = 454;
```

This example assumes that the table parts has at least three columns: price, stock, and part_id. The database will find each row with a part_id of 454. In this case, it can be assumed that part_id is a PRIMARY KEY column, so only one row will be updated. The price and stock columns of that row are then assigned new values.

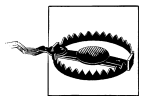
The full syntax for UPDATE can be found at UPDATE in Appendix C.

DELETE

As you might guess, the DELETE command is used to delete or remove one or more rows from a single table. The rows are completely deleted from the table:

```
DELETE FROM table_name WHERE expression;
```

The command requires only a table name and a conditional expression to pick out rows. The WHERE expression is used to select specific rows to delete, just as it is used in the UPDATE command.



If no `WHERE` condition is given, the `DELETE` command will attempt to delete *every* row of a table.

As with `UPDATE`, it is not considered an error if the `WHERE` expression evaluates to false for every row in the table, resulting in no actual deletions.

Some specific examples:

```
-- Delete the row with rowid 385:
DELETE FROM parts WHERE part_id = 385;

-- Delete all rows with a rowid greater than or equal to 43
-- and less than or equal to 246:
DELETE FROM parts WHERE part_id >= 43 AND part_id <= 246;
```

These examples assume we have a table named `parts` that contains at least one unique column named `part_id`.

As noted, if no `WHERE` clause is given, the `DELETE` command will attempt to delete every row in a table. SQLite optimizes this specific case, truncating the full table, rather than processing each individual row. Truncating the table is much faster than deleting each individual row, but truncation bypasses the individual row processing. If you wish to process each row as it is deleted, provide a `WHERE` clause that always evaluates to true:

```
DELETE FROM parts WHERE 1; -- delete all rows, force per-row processing
```

The existence of the `WHERE` clause will prevent the truncation, allowing each row to be processed in turn.

The Query Command

The final DML command to cover is the `SELECT` command. `SELECT` is used to extract or return values from the database. Almost any time you want to extract or return some kind of value, you'll need to use the `SELECT` command. Generally, the returned values are derived from the contents of the database, but `SELECT` can also be used to return the value of simple expressions. This is a great way to test out expressions, for example:

```
sqlite> SELECT 1+1, 5*32, 'abc'||'def', 1>2;
1+1         5*32        'abc' || 'def'  1>2
----------  ----------  --------------  ----------
2           160         abcdef          0
```

`SELECT` is a read-only command, and will not modify the database (unless the `SELECT` is embedded in a different command, such as a `CREATE TABLE...AS SELECT` or an `INSERT INTO...SELECT`).

Without question, `SELECT` is the most complex SQL command, both in terms of syntax as well as function. The `SELECT` syntax tries to represent a generic framework that is capable of expressing a great many different types of queries. While it is somewhat successful at this, there are areas where `SELECT` has traded away simplicity for more

flexibility. As a result, SELECT has a large number of optional clauses, each with its own set of options and formats.

Understanding how to mix and match these optional clauses to get the result you're looking for can take some time. While the most basic syntax can be shown with a good set of examples, to really wrap your head around SELECT, it is best to understand how it actually works and what it is trying to accomplish.

Because SELECT can be so complex, and because SELECT is an extremely important command, we will spend the whole next chapter looking very closely at SELECT and each of its clauses. There will be some discussion about what is going on behind the scenes, to provide more insight into how to read and write complex queries.

For now, we'll just give you a taste. That should provide enough information to play around with the other commands in this chapter. The most basic form of SELECT is:

```
SELECT output_list FROM input_table WHERE row_filter;
```

The output list is a list of expressions that should be evaluated and returned for each resulting row. Most commonly, this is simply a list of columns. The output list can also include a wildcard (*) that indicates all known columns should be returned.

The FROM clause defines the source of the table data. The next chapter will show how tables can be linked and joined, but for now we'll stick with querying one table at a time.

The WHERE clause is a conditional filtering expression that is applied to each row. It is essentially the same as the WHERE clause in the UPDATE and DELETE commands. Those rows that evaluate to true will be part of the result, while the other rows will be filtered out.

Consider this table:

```
sqlite> CREATE TABLE tbl ( a, b, c, id INTEGER PRIMARY KEY );
sqlite> INSERT INTO tbl ( a, b, c ) VALUES ( 10, 10, 10 );
sqlite> INSERT INTO tbl ( a, b, c ) VALUES ( 11, 15, 20 );
sqlite> INSERT INTO tbl ( a, b, c ) VALUES ( 12, 20, 30 );
```

We can return the whole table like this:

```
sqlite> SELECT * FROM tbl;
a           b           c           id
----------  ----------  ----------  ----------
10          10          10          1
11          15          20          2
12          20          30          3
```

We can also just return specific columns:

```
sqlite> SELECT a, c FROM tbl;
a           c
----------  ----------
10          10
11          20
12          30
```

Or specific rows:

```
sqlite> SELECT * FROM tbl WHERE id = 2;
a           b           c           id
----------  ----------  ----------  ----------
11          15          20          2
```

For more specifics, see Chapter 5 and SELECT in Appendix C.

Transaction Control Language

The Transaction Control Language is used in conjunction with the Data Manipulation Language to control the processing and exposure of changes. Transactions are a fundamental part of how relational databases protect the integrity and reliability of the data they hold. Transactions are automatically used on all DDL and DML commands.

ACID Transactions

A transaction is used to group together a series of low-level changes into a single, logical update. A transaction can be anything from updating a single value to a complex, multistep procedure that might end up inserting several rows into a number of different tables.

The classic transaction example is a database that holds account numbers and balances. If you want to transfer a balance from one account to another, that is a simple two-step process: subtract an amount from one account balance and then add the same amount to the other account balance. That process needs to be done as a single logical unit of change, and should not be broken apart. Both steps should either succeed completely, resulting in the balance being correctly transferred, or both steps should fail completely, resulting in both accounts being left unchanged. Any other outcome, where one step succeeds and the other fails, is not acceptable.

Typically a transaction is opened, or started. As individual data manipulation commands are issued, they become part of the transaction. When the logical procedure has finished, the transaction can be committed, which applies all of the changes to the permanent database record. If, for any reason, the commit fails, the transaction is rolled back, removing all traces of the changes. A transaction can also be manually rolled back.

The standard for reliable, robust transactions is the ACID test. *ACID* stands for *Atomic, Consistent, Isolated, and Durable*. Any transaction system worth using must possess these qualities.

Atomic
: A transaction should be atomic, in the sense that the change cannot be broken down into smaller pieces. When a transaction is committed to the database, the entire transaction must be applied or the entire transaction must *not* be applied. It should be impossible for only part of a transaction to be applied.

Consistent

A transaction should also keep the database consistent. A typical database has a number of rules and limits that help ensure the stored data is correct and consistent with the design of the database. Assuming a database starts in a consistent state, applying a transaction must keep the database consistent. This is important, because the database is allowed to (and is often required to) become inconsistent while the transaction is open. For example, while transferring funds, there is a moment between the subtraction from one account and the addition to another account that the total amount of funds represented in the database is altered and may become inconsistent with a recorded total. This is acceptable, as long as the transaction, as a whole, is consistent when it is committed.

Isolated

An open transaction must also be isolated from other clients. When a client opens a transaction and starts to issue individual change commands, the results of those commands are visible to the client. Those changes should *not*, however, be visible to any other system accessing the database, nor should they be integrated into the permanent database record until the entire transaction is committed. Conversely, changes committed by other clients after the transaction was started should not be visible to this transaction. Isolation is required for transactions to be atomic and consistent. If other clients could see half-applied transactions, the transactions could not claim to be atomic in nature, nor would they preserve the consistency of the database, as seen by other clients.

Durable

Last of all, a transaction must be durable. If the transaction is successfully committed, it must have become a permanent and irreversible part of the database record. Once a success status is returned, it should not matter if the process is killed, the system loses power, or the database filesystem disappears—upon restart, the committed changes should be present in the database. Conversely, if the system loses power before a transaction is committed, then upon restart the changes made within the transaction should *not* be present.

Most people think that the atomic nature of transactions is their most important quality, but all four aspects must work together to ensure the overall integrity of the database. Durability, especially, is often overlooked. SQLite tries extremely hard to guarantee that if a transaction is successfully committed, those changes are actually physically written to permanent storage and are there to stay. Compare this to traditional filesystem operations, where writes might go into an operating system file cache. Updates may sit in the cache anywhere from a few seconds to a few minutes before finally being spooled off to storage. Even then, it is possible for the data to wait around in device buffers before finally being committed to physical storage. While this type of buffering can increase efficiency, it means that a normal application really has no idea when its data is safely committed to permanent storage.

Power failures and disappearing filesystems may seem like rare occurrences, but that's not really the point. Databases are designed to deal with absolutes, especially when it comes to reliability. Besides, having a filesystem disappear is not that radical of an idea when you consider the prevalence of flash drives and USB thumb drives. Disappearing media and power failures are even more commonplace when you consider the number of SQLite databases that are found on battery-operated, handheld devices such as mobile phones and media players. The use of transactions is even more important on devices like this, since it is nearly impossible to run data recovery tools in that type of environment. These types of devices must be extremely robust and, no matter what the user does (including yanking out flash drives at inconvenient times), the system must stay consistent and reliable. Use of a transactional system can provide that kind of reliability.

Transactions are not just for writing data. Opening a transaction for an extended read-only operation is sometimes useful if you need to gather data with multiple queries. Having the transaction open keeps your view of the database consistent, ensuring that the data doesn't change between queries. That is useful if, for example, you use one query to gather a bunch of record IDs, and then issue a series of queries against each ID value. Wrapping all the queries in a transaction guarantees all of the queries see the same set of data.

SQL Transactions

Normally, SQLite is in *autocommit* mode. This means that SQLite will automatically start a transaction for each command, process the command, and (assuming no errors were generated) automatically commit the transaction. This process is transparent to the user, but it is important to realize that even individually entered commands are processed within a transaction, even if no TCL commands are used.

The autocommit mode can be disabled by explicitly opening a transaction. The `BEGIN` command is used to start or open a transaction. Once an explicit transaction has been opened, it will remain open until it is committed or rolled back. The keyword `TRANSACTION` is optional:

```
BEGIN [ DEFERRED | IMMEDIATE | EXCLUSIVE ] [TRANSACTION]
```

The optional keywords `DEFERRED`, `IMMEDIATE`, or `EXCLUSIVE` are specific to SQLite and control how the required read/write locks are acquired. If only one client is accessing the database at a time, the locking mode is largely irrelevant. When more than one client may be accessing the database, the locking mode defines how to balance peer access with ensured success of the transaction.

By default, all transactions (including autocommit transactions) use the `DEFERRED` mode. Under this mode, none of the database locks are acquired until they are required. This is the most "neighborly" mode and allows other clients to continue accessing and using the database until the transaction has no other choice but to lock them out. This allows

other clients to continue using the database, but if the locks are not available when the transaction requires them, the transaction will fail and may need to be rolled back and restarted.

`BEGIN IMMEDIATE` attempts to acquire a reserved lock immediately. If it succeeds, it guarantees the write locks will be available to the transaction when they are needed, but still allows other clients to continue to access the database for read-only operations. The `EXCLUSIVE` mode attempts to lock out *all* other clients, including read-only clients. Although the `IMMEDIATE` and `EXCLUSIVE` modes are more restrictive to other clients, the advantage is that they will fail immediately if the required locks are not available, rather than after you've issued your DDL or DML commands.

Once a transaction is open, you can continue to issue other SQL commands, including both DML and DDL commands. You can think of the changes resulting from these commands as "proposed" changes. The changes are only visible to the local client and have not been fully and permanently applied to the database. If the client process is killed or the server loses power in the middle of an open transaction, the transaction and any proposed changes it has will be lost, but the rest of the database will remain intact and consistent. It is not until the transaction is closed that the proposed changes are committed to the database and made "real." The `COMMIT` command is used to close out a transaction and commit the changes to the database. You can also use the alias `END`. As with `BEGIN`, the `TRANSACTION` keyword is optional.

```
COMMIT [TRANSACTION]
END [TRANSACTION]
```

Once a `COMMIT` has successfully returned, all the proposed changes are fully committed to the database and become visible to other clients. At that point, if the system loses power or the client process is killed, the changes will remain safely in the database.

Things don't always go right, however. Rather than committing the proposed changes, the transaction can be manually rolled back, effectively canceling the transaction and all of the changes it contains. Rolling back a set of proposed changes is useful if an error is encountered. This might be a database error, such as running out of disk space halfway through inserting a series of related records, or it might be an application logic error, such as trying to assign an invoice to an order that doesn't exist. In such cases, it usually doesn't make sense to continue with the transaction, nor does it make sense to leave inconsistent data in the database. Pretty much the only choice is to back out and try again.

To cancel the transaction and roll back all the proposed changes, you can use the `ROLLBACK` command. Again, the keyword `TRANSACTION` is optional:

```
ROLLBACK [TRANSACTION]
```

`ROLLBACK` will undo and revert all the proposed changes made by the current transaction and then close the transaction. It does not necessarily return the database to its prior state, as other clients may have been making changes in parallel. A `ROLLBACK` only cancels the proposed changes made by this client within the current transaction.

Both `COMMIT` and `ROLLBACK` will end the current transaction, putting SQLite back into autocommit mode.

Save-Points

In addition to ACID-compliant transactions, SQLite also supports *save-points*. Save-points allow you to mark specific points in the transaction. You can then accept or rollback to individual save-points without having to commit or rollback an entire transaction. Unlike transactions, you can have more than one save-point active at the same time. Save-points are sometimes called *nested transactions*.

Save-points are generally used in conjunction with large, multistep transactions, where some of the steps or sub-procedures require rollback ability. Save-points allow a transaction to proceed and (if required) roll back one step at a time. They also allow an application to explore different avenues, attempting one procedure, and if that doesn't work, trying another, without having to roll back the entire transaction to start over. In a sense, save-points can be thought of as "undo" markers in SQL command stream.

You can create a save-point with the `SAVEPOINT` command. Since multiple save-points can be defined, you must provide a name to identify the save-point:

```
SAVEPOINT savepoint_name
```

Save-points act as a stack. Whenever you create a new one, it is put at the top of the stack. Save-point identifiers do not need to be unique. If the same save-point identifier is used more than once, the one nearest to the top of the stack is used.

To release a save-point and accept all of the proposed changes made since the save-point was set, use the `RELEASE` command:

```
RELEASE [SAVEPOINT] savepoint_name
```

The `RELEASE` command does not commit any changes to disk. Rather, it flattens all of the changes in the save-point stack into the layer below the named save-point. The save-point is then removed. Any save-points contained by the named save-point are automatically released.

To cancel a set of commands and undo everything back to where a save-point was set, use the `ROLLBACK TO` command:

```
ROLLBACK [TRANSACTION] TO [SAVEPOINT] savepoint_name
```

Unlike a transaction ROLLBACK, a save-point ROLLBACK TO does not close out and eliminate the save-point. ROLLBACK TO rolls back and cancels any changes issued since the save-point was established, but leaves the transaction state exactly as it was *after* the SAVE POINT command was issued.

Consider the following series of SQL statements. The indentation is used to show the save-point stack:

```
CREATE TABLE t (i);
BEGIN;
  INSERT INTO t (i) VALUES 1;
  SAVEPOINT aaa;
    INSERT INTO t (i) VALUES 2;
    SAVEPOINT bbb;
      INSERT INTO t (i) VALUES 3;
```

At this point, if the command ROLLBACK TO bbb is issued, the state of the database would be as if the following commands were entered:

```
CREATE TABLE t (i);
BEGIN;
  INSERT INTO t (i) VALUES 1;
  SAVEPOINT aaa;
    INSERT INTO t (i) VALUES 2;
    SAVEPOINT bbb;
```

Again, notice that rolling back to save-point bbb still leaves the save-point in place. Any new commands will be associated with SAVEPOINT bbb. For example:

```
CREATE TABLE t (i);
BEGIN;
  INSERT INTO t (i) VALUES 1;
  SAVEPOINT aaa;
    INSERT INTO t (i) VALUES 2;
    SAVEPOINT bbb;
      DELETE FROM t WHERE i=1;
```

Continuing, if the command RELEASE aaa was issued, we would get the equivalent of:

```
CREATE TABLE t (i);
BEGIN;
  INSERT INTO t (i) VALUES 1;
  INSERT INTO t (i) VALUES 2;
  DELETE FROM t WHERE i=1;
```

In this case, the proposed changes from both the aaa and the enclosed bbb save-points were released and merged outward. The transaction is still open, however, and a COMMIT would still be required to make the proposed changes permanent.

Even if you have open save-points, you can still issue transaction commands. If the enclosing transaction is committed, all outstanding save-points will automatically be released and then committed. If the transaction is rolled back, all the save-points are rolled back.

If the SAVEPOINT command is issued when SQLite is in autocommit mode—that is, outside of a transaction—then a standard autocommit BEGIN DEFERRED TRANSACTION will be started. However, unlike with most commands, the autocommit transaction will not automatically commit after the SAVEPOINT command returns, leaving the system inside an open transaction. The automatic transaction will remain active until the original save-point is released, or the outer transaction is either explicitly committed or rolled back. This is the only situation when a save-point RELEASE will have a direct effect on the enclosing transaction. As with other save-points, if an autocommit save-point is rolled back, the transaction will remain open and the original save-point will be open, but empty.

System Catalogs

Many relational database systems, including SQLite, keep system state data in a series of data structures known as *system catalogs*. All of the SQLite system catalogs start with the prefix sqlite_. Although many of these catalogs contain internal data, they can be queried, using SELECT, just as if they were standard tables. Most system catalogs are read-only. If you encounter an unknown database and you're not sure what's in it, examining the system catalogs is a good place to start.

All nontemporary SQLite databases have an sqlite_master catalog. This is the master record of all database objects. If any of the tables has a populated AUTOINCREMENT column, the database will also have an sqlite_sequence catalog. This catalog is used to keep track of the next valid sequence value (for more information on AUTOINCREMENT, see "Primary keys" on page 40). If the SQL command ANALYZE has been used, it will also generate one or more sqlite_stat# tables, such as sqlite_stat1 and sqlite_stat2. These tables hold various statistics about the values and distributions in various indexes, and are used to help the query optimizer pick the more efficient query solution. For more information, see ANALYZE in Appendix C.

The most important of these system catalogs is the sqlite_master table. This catalog contains information on all the objects within a database, including the SQL used to define them. The sqlite_master table has five columns:

Column name	Column type	Meaning
type	Text	Type of database object
name	Text	Identifier name of object
tbl_name	Text	Name of associated table
rootpage	Integer	Internal use only
sql	Text	SQL used to define object

The `type` column can be `table` (including virtual tables), `index`, `view`, or `trigger`. The `name` column gives the name of the object itself, while the `tbl_name` column gives the name of the table or view the object is associated with. For tables and views, the `tbl_name` is just a copy of the `name` column. The final `sql` column holds a full copy of the original SQL command used to define the object, such as a `CREATE TABLE` or `CREATE TRIGGER` command.

Temporary databases do not have an `sqlite_master` system catalog. Rather, they have an `sqlite_temp_master` table instead.

Wrap-up

This is a long chapter packed with a huge amount of information. Even if you're familiar with a number of traditional programming languages, the declarative nature of SQL often takes time to wrap your head around. One of the best ways to learn SQL is to simply experiment. SQLite makes it easy to open up a test database, create some tables, and try things out. If you're having problems understanding the details of a command, be sure to look it up in Appendix C. In addition to more detailed descriptions, Appendix C contains detailed syntax diagrams .

If you wish to make a deeper study into SQL, there are literally hundreds of books to choose from. O'Reilly alone publishes a dozen or so titles just on the SQL language. While there are some differences between the SQL supported by SQLite and other major database systems, SQLite follows the standard fairly closely. Most of the time, SQLite deviates from the standard, it does so in an attempt to support common notations or usage in other popular database products. If you're working on wrapping you're head around some of the higher level concepts, or basic query structures, a tutorial or book written for just about any database product is likely to help. There might be a small bit of tweaking to get the queries to run under SQLite, but the changes are usually minimal.

Popular O'Reilly books covering the SQL language include *Learning SQL* (Beaulieu), *SQL in a Nutshell* (Kline, Kline, Hunt), and the *SQL Cookbook* (Molinaro). More advanced discussions can be found in *The Art of SQL* (Faroult, Robson). Popular reference books also include *SQL For Smarties* (Celko, Morgan Kaufmann) and *Introduction to SQL* (van der Lans, Addison Wesley). These two are large, but very complete. There is also *The SQL Guide to SQLite* (van der Lans, lulu.com), which takes a much deeper look at the SQL dialect specifically used by SQLite.

There are also thousands of websites and online tutorials, communities, and forums, including the SQLite mailing lists, where you can often get insightful answers to intelligent questions.

Before trying too much, be sure to read the next chapter. The next chapter is devoted to the SELECT command. In addition to covering the syntax of the command, it dives a bit deeper into what is going on behind the scenes. That foundation knowledge should make it much easier to break down and understand complex queries. It should also make it much easier to write them.