CHAPTER 4

SQL

This chapter is a complete introduction to SQL in general, and SQLite's implementation of it in particular. It assumes no previous experience with either SQL or the relational model. If you are new to SQL, SQLite should serve as an excellent springboard to entering the world of relational databases. It will give you a good grounding in the fundamentals. While the previous chapter on the relational model was a little theoretical and stuffy, this chapter is much more relaxed and practical. We're not here to prove theories; we're here to get things done. So if you didn't read the previous chapter, that's perfectly fine. You should still have no trouble with the material covered in this chapter. An understanding of the relational model is edifying, but not necessary to learn SQL.

SQL is the sole (and almost universal) means by which to communicate with a relational database. It is a language exclusively devoted to information processing. It is designed for structuring, reading, writing, sorting, filtering, protecting, calculating, generating, grouping, aggregating, and in general managing information.

SQL is an intuitive, user-friendly language. It can be fun to use and is quite powerful. One of the fun things about SQL is that regardless of whether you are an expert or a novice, it seems that you can always continue to learn new ways of doing things (for better or worse). There are often many ways to tackle a given problem, and you may find yourself taking delight in trying to find more efficient ways to get what you need, either through more compact expressions or more elegant approaches, as if solving a puzzle. And you can continue to explore dusty corners of the language you never took notice of before with which to further hone your skills, no matter your experience.

The goal of this chapter is to teach you to use SQL *well*—to expose you to good techniques, and perhaps a few cheap tricks along the way. As you can already tell, there are many different aspects to SQL. This chapter breaks them down into discrete parts and presents them in a logical order that should be relatively easy to follow. Nevertheless, SQL is a big subject and there is a lot of ground to cover. It will take some time and more than one cup of coffee to get through this chapter. But by the time you are done, you should be well equipped to put a dent in a database.

The Relational Model

As you'll remember from Chapter 3, SQL is a consequence of the relational model, which was originally proposed by E. F. Codd in 1969. The relational model requires that relational databases provide a query language, and over the years SQL rose to become the lingua franca.

The relational model consists of three essential parts: form, function, and consistency. Form refers to the structure of information. There is but one single data structure used to represent all information. This structure is called a *relation* (known in SQL as a *table*), which is made up of *tuples* (known in SQL as *rows*), which in turn are made up of *attributes* (known in SQL as *columns*).

The relational model's form is a *logical representation* of information. This logical representation is a pristine, abstract view of information unaffected by anything outside of it. It is like a mathematical concept: clean and consistent, governed by a well-defined set of deterministic rules that are not subject to change. The logical representation is completely independent of the *physical representation*, which refers to how database software stores this information on the physical level (e.g., disk). The two representations are thus distinct: nothing that occurs in the physical level can change or affect anything at the logical level. The logical level is uninhibited by hardware, software, vendor, or technology.

The second essential part of the model—the functional part—is also called the manipulative component. It defines ways of operating on information at the logical level. This was formally introduced in Codd's 1972 paper titled "Relational Completeness of Data Base Sublanguages." It added the functional part to the relational model by defining *relational algebra* and *relational calculus*. These are two formal, or "pure," query languages with a heavy basis in mathematics. Relations, as described in the data model, are mathematical sets (with a few additional properties). Relational algebra and calculus, in turn, build on this model by adding operations from set theory and formal logic, thus forming the functional component of the relational model. So both the form and function prescribed in the relational model come directly from concepts in mathematics. Each derivative, however, adds a little something to better adapt it to computers and information processing.

Query Languages

A query language connects the outside world with the abstract logical representation, and allows the two to interact. It provides a way to retrieve and modify information. It is the dynamic part of the relational model.

Codd intended relational algebra and calculus to serve as a baseline for other query languages. Relational algebra and calculus employ a highly mathematical notation that, while good for defining theory, is not terribly user-friendly in practice. Their purpose was thus to define the mathematical or relational requirements that a more user-friendly query language should support. Query languages that met these minimum requirements were called *relationally complete*. This user-friendly query language then provides a more tractable and intuitive way for a person to work with relational data, while at the same time adhering to a sound theoretical basis.

The Growth of SQL

Perhaps the first such query language that was to be so employed was that of IBM's *System R*, a relational database research project that was a direct outgrowth of Codd's papers. It was originally called SEQUEL, which stands for "Structured English Query Language." It was later shortened to SQL, or "Structured Query Language."

Other companies such as Oracle followed suit (in fact, Oracle beat IBM to market with the first SQL product), and pretty soon SQL was the de facto standard. There were other query languages, such as Ingres's QUEL, but in time SQL won out. The reasons SQL emerged as the standard may have had more to do with the dynamics of the marketplace than anything else. But the reasons why it is special today are perhaps a little clearer. There are (among others) standardization, wide adoption, and general ease of use.

SQL has been accepted as the standard language for relational databases by the American National Standards Institute (ANSI), the International Standards Organization (ISO), and the International Electrotechnical Commission (IEC). It has a well-defined standard that specifies what SQL is and does, and this standard has continued to evolve over the past two decades. To date, five versions of the standard have been published over the years (1986, 1989, 1992, 1999, and 2003). The standards are cumulative. Each version of the standard has built on the previous one, adding new features. So in effect, there is only one standard that has continued to grow and evolve over time. The first versions—SQL86 and SQL89—are collectively referred to as SQL1. SQL92 is commonly referred to as SQL2, and SQL99 as SQL3. The ANSI standard, as a whole, is very large. The SQL92 standard alone is over 600 pages. No one database product conforms to the entire standard. Nevertheless, the standard goes a long way in bringing a great deal of uniformity to relational databases.

SQL's wide adoption today is patently obvious: Oracle, Microsoft SQL Server, DB2, Informix, Sybase, PostgreSQL, MySQL, Firebird, Teradata, Intersystems Caché, and SQLite are but some of the relational databases that use SQL as their query language.

The Example Database

Before diving into syntax, let's get situated with the obligatory example database. The database used in this chapter (and the rest of this book for that matter) consists of all the foods in every episode of *Seinfeld*. If you've ever watched *Seinfeld*, you can't help but notice a slight preoccupation with food. There are more than 412 different foods mentioned in the 180 episodes of its history (according to the data I found on the Internet). That's over two new foods every show. Subtract commercial time and that's virtually a new food introduced every 10 minutes.

As it turns out, this preoccupation with food works out nicely as it makes for a database that illustrates all the requisite concepts. The database tables are shown in Figure 4-1.

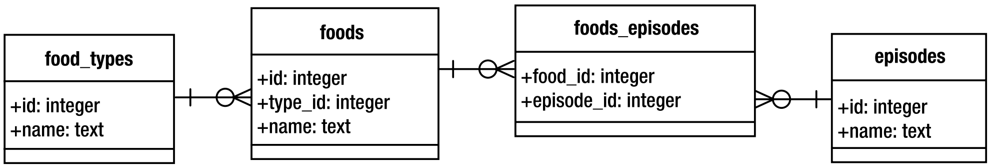


Figure 4-1. *The* Seinfeld *food database*

The database schema, as defined in SQLite, is defined as follows:

```
create table episodes (
  id integer primary key,
  season int,
  name text );
```

```
create table foods(
  id integer primary key,
  type_id integer,
  name text );

create table food_types(
  id integer primary key,
  name text );

create table foods_episodes(
  food_id integer,
  episode_id integer );
```

The main table is `foods`. Each row in `foods` corresponds to a distinct food item, the name of which is stored in the `name` attribute. The `type_id` attribute references the `food_types` table, which stores the various food classifications (e.g., baked goods, drinks, or junk food). Finally, the `foods_episodes` table links foods in `foods` with the episodes in `episodes`.

Installation

The food database is located in the examples zip file accompanying this book. It is available on the Apress website (`www.apress.com`) in the Source Code section. To create the database, first locate the `foods.sql` file in the root directory of the unpacked zip file. To create a new database from scratch from the command line, navigate to the examples directory and run the following command:

```
 sqlite3 foods.db < foods.sql
```

This will create a database file called `foods.db`. If you are not familiar with how to use the SQLite command-line program, refer to Chapter 2.

Running the Examples

For your convenience, all of the SQL examples in the chapter are available in the file `sql.sql` in the root directory of the examples zip file. So instead of typing them by hand, simply open the file and locate the SQL you want to try out.

A convenient way to run the longer queries in this chapter is to copy them into your favorite editor and save them in a separate file, which you can run from the command line. For example, copy a long query in `test.sql` to try out. You simply use the same method to run it as you did to create your database earlier:

```
sqlite3 foods.db < test.sql
```

The results will be printed to the screen. This also makes it easier to experiment with these queries without having to retype them or edit them from inside the SQLite shell. You just make your changes in the editor, save the file, and rerun the command line.

For maximum readability of the output, you may want to put the following commands at the beginning of the file:

```
.echo on
.mode col
.headers on
.nullvalue NULL
```

This causes the command-line program to 1) echo the SQL as it is executed, 2) print results in column mode, 3) include the column headers, and 4) print null values as `NULL`. The output of all examples in this chapter is formatted with these specific settings. Another option you may want to set for various examples is the `.width` option, which sets the respective column widths of the output. These vary from example to example.

For better readability, the examples are presented in two different formats. For short queries, I show the SQL and output as it would be shown from within the SQLite shell. For example:

```
sqlite> SELECT * FROM foods WHERE name='JujyFruit' AND type_id=9;

id          type_id     name
----------  ----------  ----------
244         9           JujyFruit
```

Occasionally, as in the previous example, I take the liberty of adding an extra line between the command and its output (in the shell, the output immediately follows the command). For longer queries, I show just the SQL in code format separated from the results by gray lines, as in the following example:

```
SELECT f.name name, types.name type FROM foods f
INNER JOIN (SELECT * FROM food_types WHERE id=6) types
ON f.type_id=types.id;
```

```
name                       type
-------------------------  -----
Generic (as a meal)        Dip
Good Dip                   Dip
Guacamole Dip              Dip
Hummus                     Dip
```

Syntax

SQL's declarative syntax reads a lot like a natural language. Statements are expressed in the imperative mood, beginning with the verb describing the action. Following it are the subject and predicate, as illustrated in Figure 4-2.

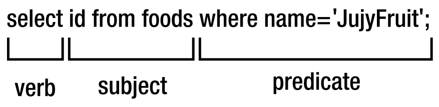


Figure 4-2. *General SQL syntax structure*

As you can see, it reads like a normal sentence. SQL was designed specifically with nontechnical people in mind, and was thus meant to be very simple and easy to understand.

Part of SQL's ease of use comes from its being (for the most part) a *declarative* language, as opposed to an imperative language such as C or Perl. A declarative language is one in which you describe *what* you want whereas an imperative language is one in which you specify *how* to get it. For example, consider the process of ordering a cheeseburger. As a customer, you use a declarative language to articulate your order. That is, you simply declare to the person behind the counter what you want:

Give me a double meat Whataburger with jalapeños and cheese, hold the mayo.

The order is passed back to chef who, on the other hand, fulfills the order using a program written in an imperative language—the recipe. He follows a series of well-defined steps that must be executed in a specific order to create the cheeseburger per your (declarative) specifications:

1. Get ground beef from the third refrigerator on the left.
2. Make a patty.
3. Cook for three minutes.
4. Flip.
5. Cook three more minutes.
6. Repeat steps 1–5 for second patty.
7. Add mustard to top bun.
8. Add patties to bottom bun.
9. Add cheese, lettuce, tomatoes, onions, and jalapeños to burger, but not mayo.
10. Combine top and bottom buns, and wrap in yellow paper.

As you can see, declarative languages tend to be more succinct than imperative ones. In this example, it took the declarative burger language (DBL) one step to materialize the cheeseburger, while it took the imperative chef language (ICL) 10 steps. Declarative languages do more with less. In fact, SQL's ease of use is not far from this example. A suitable SQL equivalent to the DBL statement above might be something along the lines of

```
SELECT burger FROM kitchen WHERE patties=2 AND toppings='jalopenos'
AND condiment != 'mayo' LIMIT 1;
```

Pretty simple. As we've mentioned, SQL was designed to be a user-friendly language. In the early days, SQL was targeted specifically for end users for tasks such as ad hoc queries and report generation (unlike today where it is almost exclusively the domain of developers and database administrators).

Commands

SQL is made up of *commands*. Commands are typically terminated by a semicolon, which marks the end of the command. For example, the following are three distinct commands:

```
SELECT id, name FROM foods;
INSERT INTO foods VALUES (NULL, 'Whataburger');
DELETE FROM foods WHERE id=413;
```

■ **Note** The semicolon, or command terminator, is associated primarily with interactive programs designed for letting users execute queries against a database. While the command terminator is commonly a semicolon, it nevertheless varies by both database system and query program. Some systems, for example, use `\g` or even the word `go`; SQLite, however, uses the semicolon as its command terminator in its command-line program and in the C API.

Commands, in turn, are composed of a series of *tokens*. Tokens can be literals, keywords, identifiers, expressions, or special characters. Tokens are separated by white space, such as spaces, tabs, and newlines.

Literals

Literals, also called constants, denote explicit values. There are three kinds: string constants, numeric constants, and binary constants. String constants are one or more alphanumeric characters surrounded by single quotes. Examples include

```
'Jerry'
'Newman'
'JujyFruit'
```

String values are delimited by single or double quotes. If single quotes are part of the string value, they must be represented as two successive single quotes. For example, Kenny's chicken would be expressed as:

```
'Kenny''s chicken'
```

Numeric constants are represented in integer, decimal, or scientific notation. Examples include

```
-1
3.142
6.0221415E23
```

Binary values are represented using the notation `x'0000'`, where each digit is a hexadecimal value. Binary values must be expressed in by multiples of 2 hexadecimal values (8 bits). Here are some examples:

```
x'01'
X'0fff'
x'0F0EFF'
X'0f0effab'
```

Keywords and Identifiers

Keywords are words that have a specific meaning in SQL. These include words like `SELECT`, `UPDATE`, `INSERT`, `CREATE`, `DROP`, and `BEGIN`. *Identifiers* refer to specific objects within the database, such as tables or indexes. Keywords are reserved words, and may not be used as identifiers. SQL is case insensitive with respect to keywords and identifiers. The following are equivalent statements:

```
SELECT * from foo;
SeLeCt * FrOm FOO;
```

Throughout this chapter, all SQL keywords are represented in uppercase and all identifiers in lowercase for clarity. By default, SQLite is case sensitive with respect to string values, so the value `'Mike'` is not the same as the value `'mike'`.

Comments

Comments in SQL are denoted by two consecutive hyphens (`--`), which comment the remaining line, or by the multiline C-style notation (`/* */`), which can span multiple lines. For example:

```
-- This is a comment on one line
/* This is a comment spanning
   two lines */
```

Creating a Database

Before you can do anything, you have to understand tables. If you don't have a table, you have nothing to work on. The table is the standard unit of information in a relational database. Everything revolves around tables. Tables are composed of rows and columns. And while that sounds simple, the sad truth is that tables are not simple. Tables bring along with them all kinds of other concepts, concepts that can't be nicely summarized in a few tidy paragraphs. In fact, it takes almost the whole chapter. So what we are going to do here is the 2-minute overview of tables—just enough for you to create a simple table, and get rid of it if you want to. And once we have that out of the way, all of the other parts of this chapter will have something to build on.

Creating Tables

Like the relational model, SQL is made up of several parts. It has a structural part, for example, which is designed to create and destroy database objects. This part of the language is generally referred to as a *data definition language* (DDL). Similarly, it has a functional part for performing operations on those objects (e.g., retrieving and manipulating data). This part of the language is referred to as a *data manipulation language* (DML). Creating tables falls under DDL, the structural part.

You create a table with the `CREATE TABLE` command, which is defined as follows:

```
CREATE [TEMP] TABLE table_name (column_definitions [, constraints]);
```

The `TEMP` or `TEMPORARY` keyword creates a *temporary table*. This kind of table is, well, temporary—it will only last as long your session. As soon as you disconnect, it will be destroyed (if you haven't already destroyed it manually). The brackets around the `TEMP` denote that it is an optional part of the command. Whenever you see any syntax in brackets, it means that the contents within them are optional. Furthermore, the pipe symbol (`|`) denotes an alternative (think of the word *or*). Take, for example, the syntax

```
CREATE [TEMP|TEMPORARY] TABLE … ;
```

This means that either the `TEMP` *or* `TEMPORARY` keyword may be optionally used. You could say `CREATE TEMP table foo...`, or `CREATE TEMPORARY TABLE foo...`. In this case, they mean the same thing.

If you don't create a temporary table, then `CREATE TABLE` creates a *base table*. The term base table refers to a table that is a named, persistent table in the database. This is the most common kind of table. There are other kinds of tables, such as *system tables* and *views*, which can exist in the database as well. But they aren't important right now. In general, the term base table is used to differentiate tables created by `CREATE TABLE` from all of the other kinds.

The minimum required information for `CREATE TABLE` is a table name and a column name. The name of the table, given by `table_name`, must be unique among all other identifiers. In the body, `column_definitions` consists of a comma-separated list of column definitions composed of a name, a *domain*, and a comma-separated list of *column constraints*. A domain, sometimes referred to as a *type*, is synonymous with a data type in a programming language. It denotes the type of information that is stored in the column. There are five native types in SQLite: `INTEGER`, `REAL`, `TEXT`, `BLOB`, and `NULL`. All of these domains are covered in the section called "Storage Classes" later in this chapter. *Constraints* are constructs that control what kind of values can be placed in the table or in individual columns. For instance, you can ensure that only unique values are placed in a column by using a `UNIQUE` constraint. Constraints are covered in the section "Data Integrity."

Following that, you can include a list of additional column constraints, denoted by `constraints`. Consider the following example:

```
CREATE TABLE contacts ( id INTEGER PRIMARY KEY,
                        name TEXT NOT NULL COLLATE NOCASE,
                        phone TEXT NOT NULL DEFAULT 'UNKNOWN',
                        UNIQUE (name,phone) );
```

Column `id` is declared to have type `INTEGER` and constraint `PRIMARY KEY`. As it turns out, the combination of this type and constraint has a special meaning in SQLite. `INTEGER PRIMARY KEY` basically turns the column into an autoincrement column (as explained in the section "Primary Key Constraints" later in this chapter). Column `name` is declared to be of type `TEXT` and has two constraints: `NOT NULL` and `COLLATE NOCASE`. Column `phone` is of type `TEXT` and has two constraints defined as well. After that, there is a table-level constraint of `UNIQUE`, which is defined for columns `name` and `phone` together.

This is a lot of information to absorb all at once, but it will all be explained in due course. I warned you that tables bring a lot of baggage with them. The only important thing here, however, is that you understand the general format of the `CREATE TABLE` statement.

Altering Tables

You can change parts of a table with the `ALTER TABLE` command. SQLite's version of `ALTER TABLE` can either rename a table or add columns. The general form of the command is

```
ALTER TABLE table { RENAME TO name | ADD COLUMN column_def }
```

Note that there is some new notation here: {}. Braces enclose a list of options, where one option is required. In this case, we have to use either `ALTER TABLE table RENAME…` or `ALTER TABLE table ADD COLUMN…`. That is, you can either rename the table using the `RENAME` clause, or add a column with the `ADD COLUMN` clause. To rename a table, you simply provide the new name given by `name`.

If you add a column, the column definition, denoted by `column_def`, follows the form in the `CREATE TABLE` statement. It is a name, followed by an optional domain and list of constraints. For example:

```
sqlite> ALTER TABLE contacts
        ADD COLUMN email TEXT NOT NULL DEFAULT '' COLLATE NOCASE;
sqlite> .schema contacts

CREATE TABLE contacts ( id INTEGER PRIMARY KEY,
                        name TEXT NOT NULL COLLATE NOCASE,
                        phone TEXT NOT NULL DEFAULT 'UNKNOWN',
                        email TEXT NOT NULL DEFAULT '' COLLATE NOCASE,
                        UNIQUE (name,phone) );
```

To view the table definition from with the SQLite command-line program, use the `.schema` shell command followed by the table name. It will print the current table definition. If you don't provide a table name, then `.schema` will print the entire database schema.

Tables can also be created from `SELECT` statements, allowing you to create not only the structure but also the data at the same time. This particular use of the `CREATE TABLE` statement is covered later in the section "Inserting Records."

Querying the Database

As mentioned previously in the section "Creating a Database," the manipulative component of SQL is called Data Manipulation Language (DML). DML, in turn has two basic parts: data retrieval and data modification. Data retrieval is the domain of the `SELECT` command. It is the sole command for querying the database. `SELECT` is by far the largest and most complex command in SQL. `SELECT` derives many of its operations from relational algebra, and encompasses a large portion of it.

Relational Operations

There are 13 relational operations used in `SELECT`, which are divided into three categories:

- Fundamental Operations
  - Restriction
  - Projection
  - Cartesian Product
  - Union
  - Difference
  - Rename
- Additional Operations
  - Intersection
  - Natural Join
  - Assign
- Extended Operations
  - Generalized Projection
  - Left Outer Join
  - Right Outer Join
  - Full Outer Join

The fundamental operations are just that: fundamental. They define the basic relational operations, and all of them (with the exception of *rename*) have their basis in set theory. The additional operations are for convenience. They can be expressed (or performed) in terms of the fundamental operations. They simply offer a shorthand way of performing frequently used combinations of the fundamental operations. Finally, the extended operations add features to the fundamental and additional operations. The generalized projection operation adds arithmetic expressions, aggregates, and grouping features to the fundamental projection operations. The outer joins extend the join operations and allow additional information and/or incomplete information to be retrieved from the database.

In ANSI SQL, `SELECT` can perform every one of these relational operations. These operations in turn make up all of the original relational operators defined by Codd (and then some) with one exception—*divide*. SQLite supports all of the relational operations in ANSI SQL with the exception of right and full outer joins (although these operations can be performed by other indirect means).

All of these operations are defined in terms of relations. They take one or more relations as their inputs, and produce a relation as their output. It means that the fundamental structure of information never changes as a result of the operations performed upon it. Relational operations take only relations and produce only relations. This property is called *closure*. Closure makes possible *relational expressions*. The output of one operation, being a relation, can therefore serve as the input to another operation. Thus, operations can take as their arguments not only relations but also other operations (as they will produce relations). This allows operations to be strung together into relational expressions. Relational expressions can therefore be created to arbitrary complexity. For example, the output of a SELECT operation (a relation) can be fed as the input to another, as follows:

```
SELECT name FROM (SELECT name, type_id FROM (SELECT * FROM foods));
```

Here, the output of the innermost SELECT is fed to the next SELECT, whose output is in turn fed to the outermost SELECT. It is all a single relational expression.

The end result is that relational operations are not only powerful in their own right but can be combined to make even more powerful and elaborate expressions. As it turns out, the additional operators, as mentioned earlier, are in fact relational expressions composed of fundamental operations. For example, the intersection operation is defined in terms of two *difference* operations.

The Operational Pipeline

Syntactically, the SELECT command incorporates many of the relational operations through a series of *clauses*. Each clause corresponds to specific relational operation. Almost all of the clauses are optional, so you can selectively employ only the operations you need to obtain the data you are looking for.

SELECT is a very large command. A very general form of SELECT (without too much distracting syntax) can be represented as follows:

```
SELECT DISTINCT heading FROM tables WHERE predicate
       GROUP BY columns HAVING predicate
       ORDER BY columns LIMIT count,offset;
```

Each keyword—DISTINCT, FROM, WHERE, HAVING—is a separate clause. Each clause is made up of the keyword (represented in uppercase) followed by arguments (represented in italics). The syntax of the particular arguments varies according to the clause. The clauses, their corresponding relational operations, and their various arguments are listed in Table 4-1. The ORDER BY clause does not correspond to a formal relational operation as it only reorders rows—that is the meaning of the double hyphens in the Operation column. From here on out, I will in most cases refer to these clauses simply by name. That is, rather than "the WHERE clause," I will simply use WHERE. While there is both a SELECT command and a SELECT clause, I will refer to the command as the SELECT command, and the clause by the convention.

Table 4-1. *SELECT Clauses*

Order	Clause	Operation	Input
1	FROM	Join	List of tables
2	WHERE	Restriction	Logical predicate
3	ORDER BY	--	List of columns
4	GROUP BY	Restriction	List of columns
5	HAVING	Restriction	Logical predicate
6	SELECT	Projection	List of columns or expressions
7	DISTINCT	Restriction	List of columns
8	LIMIT	Restriction	Integer value
9	OFFSET	Restriction	Integer value

Operationally, the SELECT command is like a pipeline that processes relations. This pipeline has optional processes that you can plug into it as you need them. Regardless of whether you include or exclude a particular operation, the relative operations is always the same. This order is shown in Figure 4-3.

The SELECT command starts with FROM, which takes one or more input relations, combines them into a single composite relation, and passes it through the subsequent chain of operations. Each subsequent operation takes exactly one relation as input and produces exactly one relation as output.

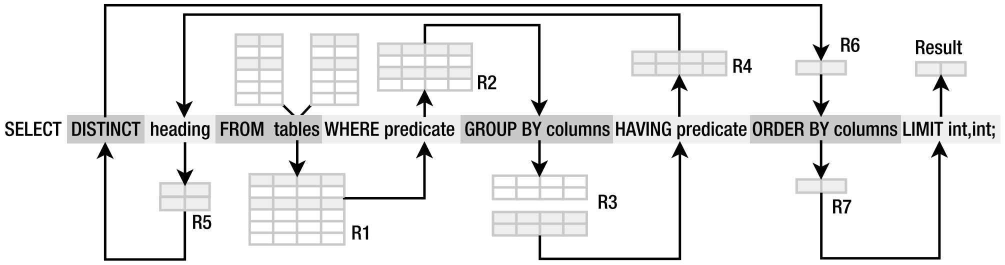

SELECT DISTINCT heading FROM tables WHERE predicate GROUP BY columns HAVING predicate ORDER BY columns LIMIT int,int;
R1
R2
R3
R4
R5
R6
R7
Result

Figure 4-3. *SELECT phases*

All operations are optional with the exception of SELECT. You must always provide at least this clause to make a valid SELECT command. By far the most common invocation of the SELECT command consists of three clauses: SELECT, FROM, and WHERE. This basic syntax and its associated clauses are shown as follows:

```
SELECT heading FROM tables WHERE predicate;
```

The FROM clause is a comma-separated list of one or more tables (represented by the variable tables in Figure 4-3). If more than one table is specified, they will be combined to form a single relation (represented by *R1* in Figure 4-3). This is done by one of the join operations. The resulting relation produced by FROM serves as the initial material. All subsequent operations will either work directly from it or from derivatives of it.

The WHERE clause filters rows in *R1*. In a way, it determines the number of rows (or *cardinality*) of the result. WHERE is a restriction operation, which (somewhat confusingly) is also known as *selection*. Restriction takes a relation and produces a row-wise subset of that relation. The argument of WHERE is a *predicate*, or logical expression, which defines the selection criteria by which rows in *R1* are included in (or excluded from) the result. The selected rows from the WHERE clause form a new relation *R2*, as shown in Figure 4-3. *R2* is a restriction of *R1*.

In this particular example, *R2* passes through the other operations unscathed until it reaches the SELECT clause. The SELECT clause filters the columns in *R2*. Its argument consists of a comma-separated list of columns or expressions that define the result. This is called the *projection list*, or *heading* of the result. The number of columns in the heading is called the *degree* of the result. The SELECT clause is a projection operation, which is the process of producing a column-wise subset of a relation. Figure 4-4 illustrates a projection operation. The input is a relation composed of five columns, including the columns 1, 2, and 3. The projection operation as represented by the arrow produces a new relation by extracting these columns and discarding the others.

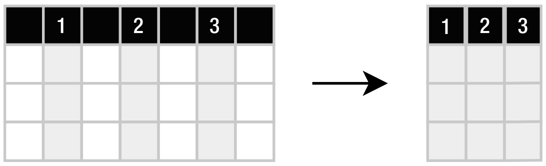


Figure 4-4. *Projection*

Consider the following example:

```
sqlite> SELECT id, name FROM food_types;
id          name
----------  ----------
1           Bakery
2           Cereal
3           Chicken/Fowl
4           Condiments
5           Dairy
6           Dip
7           Drinks
8           Fruit
9           Junkfood
10          Meat
11          Rice/Pasta
12          Sandwiches
13          Seafood
14          Soup
15          Vegetables
```

There is no `WHERE` clause to filter rows. The `SELECT` clause specifies all of the columns in `food_types`, and the `FROM` clause does not join tables. So there really isn't much taking place. The result is an exact copy of `food_types`. The input relation and the result relation are the same. If you want to include all possible columns in the result, rather than listing them one by one, you can use a shorthand notation—an asterisk (`*`)—instead. Thus, the previous example can just as easily be expressed as

```
SELECT * FROM food_types;
```

This is almost the simplest `SELECT` command. In reality, the simplest `SELECT` command requires only a `SELECT` clause, and the `SELECT` clause requires only a single argument. Therefore, the simplest `SELECT` command is

```
sqlite> SELECT NULL;

NULL
----
NULL
```

The result is a relation composed of one row and one column, with an unknown value—`NULL`. `NULL` is covered later in the section "The Thing Called Null."

Now let's look at an example that actually does something operationally. Consider the following:

```
SELECT name, type_id FROM foods;
```

This performs a projection on `foods`, selecting two of its three columns—`name` and `type_id`. The `id` column is thrown out, as shown in Figure 4-5.

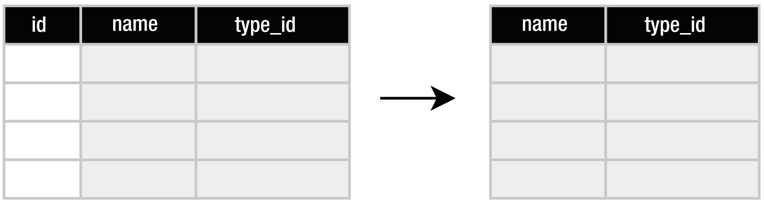


Figure 4-5. *A projection of foods*

Let's summarize: the `FROM` clause takes the input relations and performs a join, which combines them into a single relation *R1*. The `WHERE` clause takes *R1* and filters it via restriction, producing a new relation *R2*. The `SELECT` clause takes *R2* and performs projection, producing the final result. This process is shown in Figure 4-6.

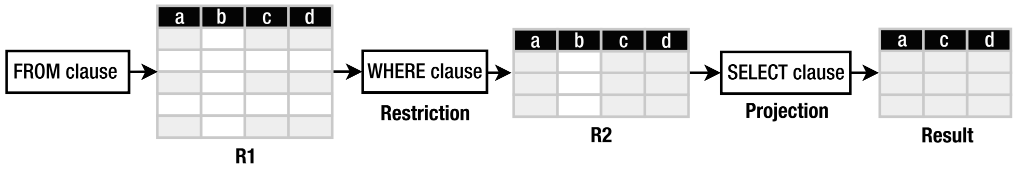


Figure 4-6. *Restriction and projection in SELECT*

With this simple example, you can begin to see how a query language in general and SQL in particular ultimately operates in terms of relational operations. There is real math under the hood.

Filtering

If the `SELECT` command is the most complex command in SQL, then the `WHERE` clause is the most complex clause in `SELECT`. And, just as the `SELECT` command pulls in aspects of set theory, the `WHERE` clause also pulls in aspects of formal logic. By and large, the `WHERE` clause is usually the part of the `SELECT` command that harbors the most complexity. But it also does most of the work. Having a solid understanding of its mechanics will most likely bring the best overall returns in your day-to-day use of SQL.

The database applies the `WHERE` clause to each row of the relation produced by the `FROM` clause (*R1*). As stated earlier, `WHERE`—a restriction—is a filter. The argument of `WHERE` is a logical predicate. A predicate, in the simplest sense, is just an assertion about something. Consider the following statement:

The dog (subject) is purple and has a toothy grin (predicate).

The dog is the subject and the predicate consists of the two assertions (color is purple and grin is toothy). This statement may be true or false, depending on the dog the predicate is applied to. In the terminology of formal logic, a statement consisting of a subject and a predicate is called a *proposition*. All propositions are either true or false.

A predicate then says something about a subject. The subject in the `WHERE` clause is a row. The row is the logical subject. The `WHERE` clause is the logical predicate. Together (as in a grammatical sentence) they form a logical proposition, which evaluates to true or false. This proposition is formulated and evaluated for every row in *R1*. Each row in which the proposition evaluates to true is included (or selected) as part of the result (*R2*). Each row in which it is false is excluded. So the dog proposition translated into a relational equivalent would look something like this:

```
SELECT * FROM dogs WHERE color='purple' AND grin='toothy';
```

The database will take each row in relation `dogs` (the subject) and apply the `WHERE` clause (the predicate) to form the logical proposition:

```
This row has color='purple' AND grin='toothy'.
```

This is either true of the given row, or it is false—nothing more. If it is true, then the row (or dog) is indeed purple and toothy, and it is included in the result. If the proposition is false, then the row may be purple but not toothy, or toothy but not purple, or neither purple nor toothy. In any case, the proposition is false and the row is therefore excluded.

`WHERE` is a powerful filter. It provides you with a great degree of control over the conditions with which to include (or exclude) rows in (or from) the result. As a logical predicate, it is also a logical expression. A logical expression is an expression that evaluates to exactly one of two possible logical outcomes: *true or false*. A logical predicate then is just a logical expression that is used in a specific way—to qualify a subject and form a proposition.

At their simplest, logical expressions consist of two *values* or *value expressions* compared by *relational operators*. *Value expressions* are built from *values* and *operators*.

A relational operator referred to here is a relational operator in the mathematical sense. It is different from a relational operator in the relational sense, as defined in relational algebra. A relational operator in math is an operator that relates two values and evaluates to true or false (e.g., $x>y$, $x<=y$). A relational operator in relational algebra is an operator that takes two or more relations and produces a new relation. To minimize confusion, the scope of this discussion is limited to one relational operation—restriction—as implemented by the `WHERE` clause in SQL. The `WHERE` clause is expressed in terms of logical propositions, which use relational (in the mathematical sense) operators. For the remainder of this section, "relational operator" refers specifically to that in the mathematical sense.

Values

Everything begins with *values*, which represent some kind of data in the real world. Values can be classified by their domain (or type), such as a numerical value (1, 2, 3, etc.) or string value ("JujyFruit"). Values can be expressed as *literal values* (an explicit quantity such as 1, 2, 3 or "JujyFruit"), *variables* (often in the form of column names like `foods.name`), expressions (3+2/5), or the results of functions (`COUNT(foods.name)`)—which are covered later.

Operators

An operator takes one or more values as input and produces a value as output. An operator is so named because it performs some kind of operation, producing some kind of result. *Binary operators* are operators that take two input values (or operands). *Ternary operators* take three operands, *unary operators* take just one, and so on.

Many operators produce the same kind of information they consume (operators that operate on numbers produce numbers, etc.). Such operators can be strung together, feeding the output of one operator into the input of another (Figure 4-7), forming value expressions.

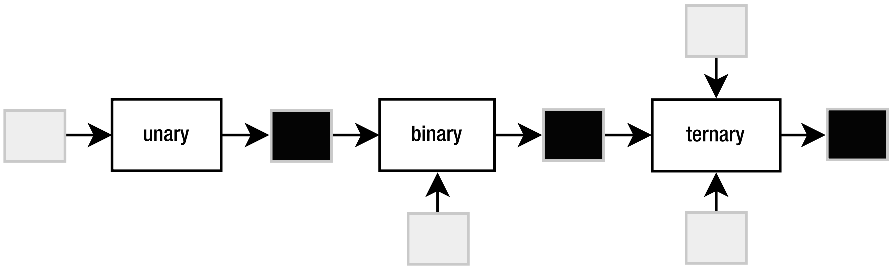


Figure 4-7. *Unary, binary, and ternary operators*

By stringing operators together, you can create value expressions that are expressed in terms of yet other value expressions to arbitrary complexity. For example:

```
x = count(episodes.name)
y = count(foods.name)
z = y/x * 11
```

Binary Operators

Binary operators are by far the most common of all operators in SQL. Table 4-2 lists the binary operators supported in SQLite by *precedence*, from highest to lowest. Operators in each color group have equal precedence. Precedence determines the default order of evaluation in an expression with multiple operators. For example, take the expression 4+3*7. It evaluates to 25. The multiplication operator has higher precedence than addition, and is therefore evaluated first. So the expression is computed (3*7)+4. Precedence can be overridden using parentheses. The expression (4+3)*7 is not 25. Here, the parentheses have declared an explicit order of operation.

Table 4-2. *Binary Operators*

Operator	Type	Action
`||`	String	Concatenation
*	Arithmetic	Multiply
/	Arithmetic	Divide
%	Arithmetic	Modulus
+	Arithmetic	Add

Table 4-2. *Binary Operators (Continued)*

Operator	Type	Action
–	Arithmetic	Subtract
<<	Bitwise	Right shift
>>	Bitwise	Left shift
&	Logical	And
|	Logical	Or
<	Relational	Less than
<=	Relational	Less than or equal to
>	Relational	Greater than
>=	Relational	Greater than or equal to
=	Relational	Equal to
==	Relational	Equal to
<>	Relational	Not equal to
!=	Relational	Not equal to
IN	Logical	In
AND	Logical	And
OR	Logical	Or
LIKE	Relational	String matching
GLOB	Relational	Filename matching

Arithmetic operators (e.g., addition, subtraction, division) are binary operators that take numeric values and produce a numeric value. *Relational operators* (e.g., >, <, =) are binary operators that compare values and value expressions and return a *logical value* (also called a *truth* value), which is either true or false. Relational operators form *logical expressions*, for example:

```
x > 5
1 < 2
```

A logical expression is any expression that returns a truth value. In SQLite, false is represented by the number 0, while true is represented by anything else. For example:

```
sqlite> SELECT 1 > 2;

1 > 2
----------
0
```

```
sqlite> SELECT 1 < 2;

1 < 2
----------
1

sqlite> SELECT 1 = 2;

1 = 2
----------
0

sqlite> SELECT -1 AND 1;

-1 AND 1
----------
1
```

Logical Operators

Logical operators (AND, OR, NOT, IN) are binary operators that operate on truth values or logical expressions. They produce a specific truth value depending on their inputs. They are used to build more complex logical expressions from simpler expressions, such as

```
(x > 5) AND (x != 3)
(y < 2) OR (y > 4) AND NOT (y = 0)
(color='purple') AND (grin='toothy')
```

The truth value produced by a logical operator for a given pair of arguments depends on the operator. For example, logical AND requires that both input values evaluate to true in order for it to return *true*. Logical OR, on the other hand, only requires that one input value evaluate to *true* in order for it to return *true*. All possible outcomes for a given logical operator are defined in what is known as a *truth table*. The truth tables for AND and OR are shown in Tables 4-3 and 4-4, respectively.

Table 4-3. *Truth Table for Logical And*

Argument 1	Argument 2	Result
True	True	True
True	False	False
False	False	False

Table 4-4. *Truth Table for Logical Or*

Argument 1	Argument 2	Result
True	True	True
True	False	True
False	False	False

This is the stuff the WHERE clause is made of. Using logical operators, you can create a complex logical predicate. The predicate is what defines how WHERE's relational restriction operation restricts. For example:

```
sqlite> SELECT * FROM foods WHERE name='JujyFruit' AND type_id=9;

id          type_id     name
----------  ----------  ----------
244         9           JujyFruit
```

The restriction here works according to the expression (name='JujyFruit') AND (type_id=9), which consists of two logical expressions joined by logical AND. Both of these conditions must be true for any record in foods to be included in the result.

The LIKE Operator

A particularly useful relational operator is LIKE. LIKE is similar to equals (=), but is used for matching string values against patterns. For example, to select all rows in foods whose names begin with the letter "J," you could do the following:

```
sqlite> SELECT id, name FROM foods WHERE name LIKE 'J%';

id     name
-----  --------------------
156    Juice box
236    Juicy Fruit Gum
243    Jello with Bananas
244    JujyFruit
245    Junior Mints
370    Jambalaya
```

A percent symbol (%) in the pattern matches any sequence of zero or more characters in the string. An underscore (_) in the pattern matches any single character in the string. The percent symbol is greedy. It will eat everything between two characters except those characters. If it is on the extreme left or right of a pattern, it will consume everything on each respective side. Consider the following examples:

```
sqlite> SELECT id, name FROM foods WHERE name LIKE '%ac%P%';

id     name
-----  --------------------
127    Guacamole Dip
168    Peach Schnapps
198    Mackinaw Peaches
```

Another useful trick is to use NOT to negate a pattern:

```
sqlite> SELECT id, name FROM foods
        WHERE name like '%ac%P%' AND name NOT LIKE '%Sch%'

id     name
-----  --------------------
 38    Pie (Blackberry) Pie
127    Guacamole Dip
198    Mackinaw peaches
```

Limiting and Ordering

You can limit the size and particular range of the result using the LIMIT and OFFSET keywords. LIMIT specifies the maximum number of records to return. OFFSET specifies the number of records to skip. For example, the following statement obtains the Cereal record (the second record in food_types) using LIMIT and OFFSET:

```
SELECT * FROM food_types LIMIT 1 OFFSET 1 ORDER BY id;
```

The OFFSET clause skips one row (the Bakery row) and the LIMIT clause returns a maximum of one row (the Cereal row).

But there is something else here as well: ORDER BY. This clause sorts the result by a column or columns before it is returned. The reason it is important in this example is because the rows returned from SELECT are never guaranteed to be in any specific order—the SQL standard declares this. Thus, the ORDER BY clause is essential if you need to count on the result being in any specific order. The syntax of the ORDER BY clause is similar to the SELECT clause: it is a comma-separated list of columns. Each entry may be qualified with a sort order—ASC (ascending, the default) or DESC (descending). For example:

```
sqlite> SELECT * FROM foods WHERE name LIKE 'B%'
        ORDER BY type_id DESC, name LIMIT 10;

id     type_id   name
-----  --------  --------------------
382    15        Baked Beans
383    15        Baked Potato w/Sour
384    15        Big Salad
385    15        Broccoli
362    14        Bouillabaisse
328    12        BLT
```

```
327    12        Bacon Club (no turke
326    12        Bologna
329    12        Brisket Sandwich
274    10        Bacon
```

Typically you only need to order by a second (third, etc.) column when there are duplicate values in the first (second, etc.) ordered column(s). Here, there were many duplicate `type_id`s. I wanted to group them together, and then arrange the foods alphabetically within these groups.

■ **Note** `LIMIT` and `OFFSET` are not standard SQL keywords as defined in the ANSI standard. Nevertheless, they are found on several other databases, such as MySQL and PostgreSQL. Oracle, MS SQL, and Firebird also have functional equivalents, although they use different syntax.

If you use both `LIMIT` and `OFFSET` together, you can use a comma notation in place of the `OFFSET` keyword. For example, the following SQL

```
SELECT * FROM foods WHERE name LIKE 'B%'
ORDER BY type_id DESC, name LIMIT 1 OFFSET 2;
```

can be expressed equivalently with

```
sqlite> SELECT * FROM foods WHERE name LIKE 'B%'
        ORDER BY type_id DESC, name LIMIT 1,2;

id     type_id   name
-----  --------  --------------------
384    15        Big Salad
```

Here, the comma following `LIMIT 1` adds the `OFFSET` of 2 to the clause. Also, note that `OFFSET` depends on `LIMIT`. That is, you can use `LIMIT` without using `OFFSET` but not the other way around.

Notice that `LIMIT` and `OFFSET` are dead last in the operational pipeline. One common misconception of `LIMIT/OFFSET` is that it speeds up a query by limiting the number of rows that must be collected by the `WHERE` clause. This is not true. If it were, then `ORDER BY` would not work properly. For `ORDER BY` to do its job, it must have the entire result in hand to provide the correct order.

`ORDER BY`, on the other hand, works after `WHERE` but before `SELECT`. How do I know this? The following statement works:

```
SELECT name FROM foods ORDER BY id;
```

I am asking SQLite to order by a column that is not in the result. The only way this could happen is if the ordering takes place before projection (while the `id` column is still in the set). While this works in SQLite, it is also specified in SQL2003.

Functions and Aggregates

Relational algebra supports the notion of functions and aggregates through the extended operation known as *generalized projection*. The `SELECT` clause is a generalized projection rather than just a fundamental projection. The fundamental projection operation only accepts column names in the projection list as a means to produce a column-wise subset. Generalized projection accepts this as well as arithmetic expressions, functions, and aggregates in the projection list, in addition to other features such as `GROUP BY` and `HAVING`, all of which are covered here and in the subsequent sections.

SQLite comes with various built-in functions and aggregates that can be used within various clauses. Function types range from mathematical functions such as `ABS()`, which computes the absolute value, to string formatting functions such as `UPPER()` and `LOWER()`, which convert text to upper- and lowercase, respectively. For example:

```
sqlite> SELECT UPPER('hello newman'), LENGTH('hello newman'), ABS(-12);

UPPER('hello newman')  LENGTH('hello newman') ABS(-12)
---------------------  --------------------   ----------
HELLO NEWMAN           12                     12
```

Notice that the function names are case insensitive (i.e., `upper()` and `UPPER()` refer to the same function). Functions can accept column values as their arguments:

```
sqlite> SELECT id, UPPER(name), LENGTH(name) FROM foods
        WHERE type_id=1 LIMIT 10;

id     UPPER(name)                 LENGTH(name)
-----  --------------------------  ------------
1      BAGELS                      6
2      BAGELS, RAISIN              14
3      BAVARIAN CREAM PIE          18
4      BEAR CLAWS                  10
5      BLACK AND WHITE COOKIES     23
6      BREAD (WITH NUTS)           17
7      BUTTERFINGERS               13
8      CARROT CAKE                 11
9      CHIPS AHOY COOKIES          18
10     CHOCOLATE BOBKA             15
```

Since functions can be a part of any expression, they can also be used in the `WHERE` clause:

```
sqlite> SELECT id, UPPER(name), LENGTH(name) FROM foods
        WHERE LENGTH(name) < 5 LIMIT 5;

id     upper(name)           length(name)
-----  --------------------  --------------------
36     PIE                   3
48     BRAN                  4
56     KIX                   3
57     LIFE                  4
80     DUCK                  4
```

Just for reinforcement, let's go through the relational operations performed to carry out the preceding statement:

1. `FROM` clause (join)

   The `FROM` clause in this case does not join tables; it only produces a relation *R1* containing all the rows in `foods`.

2. `WHERE` clause (restriction)

   For each row in *R1*:

   a. Apply the predicate `LENGTH(name) < 5` to the row. That is, evaluate the proposition "row has LENGTH(name) < 5."

   b. If true, add the row to *R2*.

3. `SELECT` clause (projection)

   For each row in *R2*:

   a. Create a new row *r* in *R3*.

   b. Copy the value of the `id` field in restriction into the first column of *r*.

   c. Copy the result of the expression `UPPER(row.name)` to the second column of *r*.

   d. Copy the result of the expression `LENGTH(row.name)` to the third column or *r*.

4. `LIMIT` clause (restriction)

   Restrict *R3* to just the first five records.

Aggregates are a special class of functions that calculate a composite (or aggregate) value over a group of rows (or relation). According to Webster, an aggregate, by definition, is a value "formed by the collection of units or particles into a body, mass, or amount." The particles here are rows in a table. Standard aggregate functions include `SUM()`, `AVG()`, `COUNT()`, `MIN()`, and `MAX()`. For example, to get a count of the number of foods that are baked goods (`type_id=1`), we can use the `COUNT` aggregate as follows:

```
sqlite> SELECT COUNT(*) FROM foods WHERE type_id=1;

count
-----
47
```

The `COUNT` aggregate returns a count of every row in the relation. Whenever you see an aggregate, you should automatically think, "For each row in a table, do something." It is the computed value obtained from doing something with each row in the table. For example, `COUNT` might be expressed in terms of the following pseudocode:

```
int COUNT():
    count = 0;
    for each row in Relation:
        count = count + 1
    return count;
```

Aggregates can aggregate not only column values, but any expression—including functions. For example, to get the average length of all food names, you can apply the AVG aggregate to the LENGTH(name) expression as follows:

```
sqlite> SELECT AVG(LENGTH(name)) FROM foods;

AVG(LENGTH(name))
-----------------
12.58
```

Aggregates operate within the SELECT clause. They compute their values on the rows selected by the WHERE clause—not from all rows selected by the FROM clause. The SELECT command filters first, then aggregates.

While SQLite comes with a standard set of common SQL functions and aggregates, it is worth noting that the SQLite C API allows you to create custom functions and aggregates as well. See Chapter 7 for more information. For reference, a complete list of the built-in functions and aggregates in SQLite can be found in Appendix A.

Grouping

An essential part of aggregation is grouping. That is, in addition to computing aggregates over an entire result, you can also split that result into groups of rows with like values, and compute aggregates on each group—all in one step. This is the job of the GROUP BY clause. For example:

```
sqlite> SELECT type_id FROM foods GROUP BY type_id;

type_id
----------
1
2
3
.
.
.
15
```

GROUP BY is a bit different than the other parts of SELECT, so you need to use your imagination a little to wrap your head around it. The process is illustrated in Figure 4-8. Operationally, GROUP BY sits in between the WHERE clause and the SELECT clause. GROUP BY takes the output of WHERE and splits it into groups of rows that share a common value (or values) for a specific column (or columns). These groups are then passed to the SELECT clause. In the example, there are 15 different food types (type_id ranges from 1 to 15), and therefore GROUP BY organizes all rows in foods into 15 groups varying by type_id. SELECT takes each group and extracts its common type_id value and puts it into a separate row. Thus, there are 15 rows in the result, one for each group.

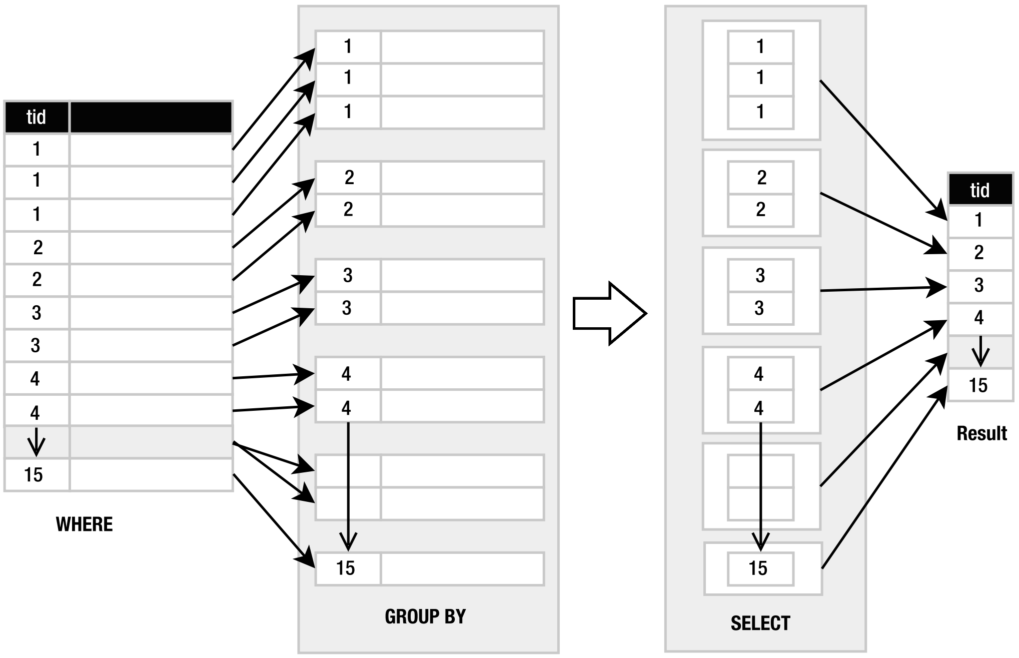


Figure 4-8. *GROUP BY process*

When GROUP BY is used, the SELECT clause applies aggregates to each group separately, rather than the entire result as a whole. Since aggregates produce a single value from a group of values, they collapse these groups of rows into single rows. For example, consider applying the COUNT aggregate to the preceding example to get the number of records in each type_id group:

```
sqlite> SELECT type_id, COUNT(*) FROM foods GROUP BY type_id;

type_id     COUNT(*)
----------  ----------
1           47
2           15
3           23
4           22
5           17
6           4
7           60
8           23
9           61
10          36
11          16
12          23
13          14
14          19
15          32
```

Here, COUNT() was applied 15 times—once for each group, as illustrated in Figure 4-9. Note that in the diagram the actual number of records in each group is not represented literally (e.g., it doesn't show 47 records in the group for type_id=1).

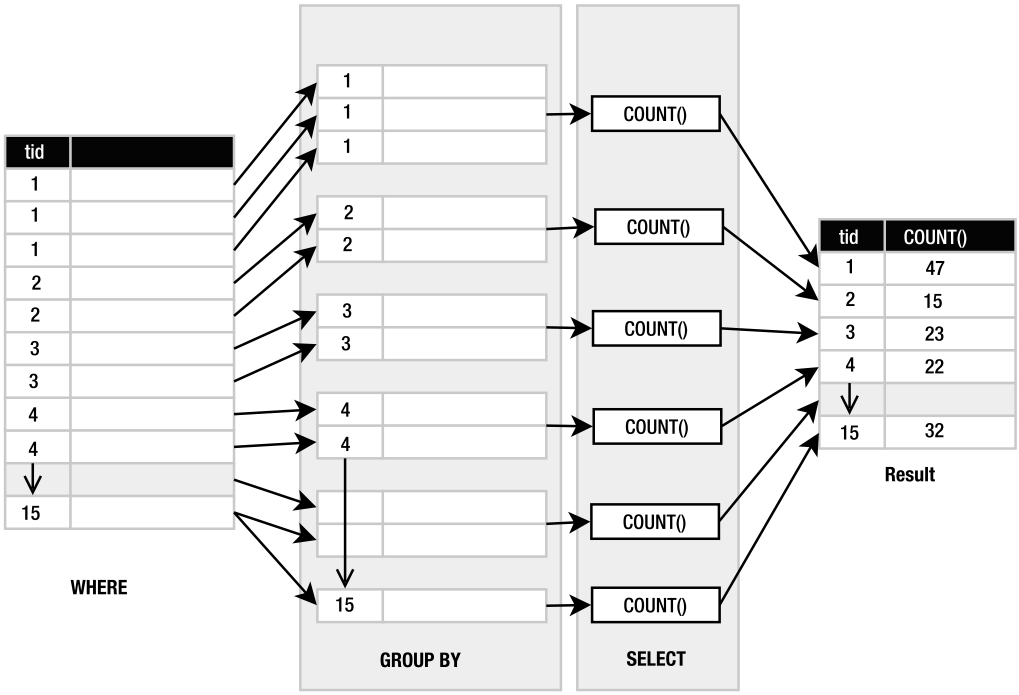


Figure 4-9. *GROUP BY and aggregation*

The number of records with type_id=1 (Baked Goods) is 47. The number with type_id=2 (Cereal) is 15. The number with type_id=3 (Chicken/Fowl) is 23, and so forth. So, to get this information, you could run 15 queries as follows:

```
select count(*) from foods where type_id=1;
select count(*) from foods where type_id=2;
select count(*) from foods where type_id=3;
.
.
.
select count(*) from foods where type_id=15;
```

Or, you get the results using the single SELECT with a GROUP BY as follows:

```
SELECT type_id, COUNT(*) FROM foods GROUP BY type_id;
```

But there is more. Since GROUP BY has to do all this work to create groups with like values, it seems a pity not to let you filter these groups before handing them off to the SELECT clause. That is the purpose of HAVING, a predicate that you apply to the result of GROUP BY. It filters the groups from GROUP BY in the same way that the WHERE clause filters rows from the FROM clause.

The only difference is that the WHERE clause's predicate is expressed in terms of individual row values, and HAVING's predicate is expressed in terms of aggregate values.

Take the previous example, but this time say you are only interested in looking at the food groups that have fewer than 20 foods in them:

```
sqlite> SELECT type_id, COUNT(*) FROM foods
        GROUP BY type_id HAVING COUNT(*) < 20;

type_id     COUNT(*)
----------  ----------
2           15
5           17
6           4
11          16
13          14
14          19
```

Here, HAVING applies the predicate COUNT(*)<20 to all of the groups. Any group that does not satisfy this condition (that has 20 or more foods in it) is not passed on to the SELECT clause. Figure 4-10 illustrates this restriction.

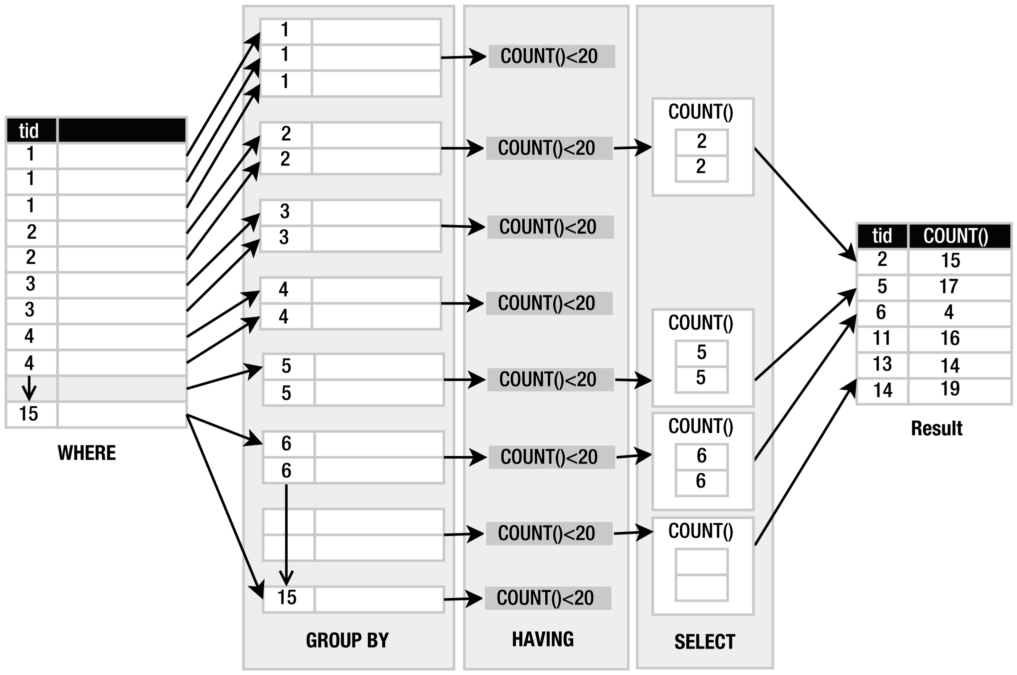


Figure 4-10. *HAVING as group restriction*

The third column in the figure shows groups of rows ordered by type_id. The values shown are the type_id values. The actual number of rows shown in each group is not exact, but figurative. I have only shown two rows in each group to represent the group as a whole.

So GROUP BY and HAVING work as additional restriction phases. GROUP BY takes the restriction produced by the WHERE clause and breaks it into groups of rows that share a common value for a given column. HAVING then applies a filter to each of these groups. The groups that make it through are passed on to the SELECT clause for aggregation and projection.

Removing Duplicates

The next operation in the pipeline is yet another restriction: DISTINCT. DISTINCT takes the result of the SELECT clause and filters out duplicate rows. For example, you'd use this to get all distinct type_id values from foods:

```
sqlite> SELECT DISTINCT type_id FROM foods;

type_id
----------
1
2
3
.
.
.
15
```

This statement works as follows: the WHERE clause returns the entire foods table (all 412 records). The SELECT clause pulls out just the type_id column, and finally DISTINCT removes duplicate rows, reducing the number from 412 rows to 15 rows, all unique. This particular example produces the same result as the GROUP BY example, but it goes about it in a completely different manner. DISTINCT simply compares all columns of all rows listed in the SELECT clause and removes duplicates. There is no grouping on a particular column or columns, nor do you use any predicates. DISTINCT is just a uniqueness filter.

Joining Tables

Your current knowledge of SELECT so far is based entirely on its filtering capabilities: start with something, remove rows, remove columns, aggregate, remove duplicates, and perhaps limit the number of rows even more. The SELECT, WHERE, GROUP BY, HAVING, DISTINCT, and LIMIT clauses are all filters of some sort.

Filtering, however, is only half of the picture. SELECT has two essential parts: collect and refine. Up until now you've only seen the refining, or filtering, part. The source of all this refinement has been only a single table. But SELECT is not limited to filtering just a single table; it can link tables together. SELECT can construct a larger and more detailed picture made of different parts of different tables, and treat that composite as your input table. Then you apply the various filters to whittle it down and isolate the parts you're interested in. This process of linking tables together is called *joining*. Joining is the work of the FROM clause.

Joins are the first operation(s) of the SELECT command. They produce the initial information to be filtered and processed by the remaining parts of the statement. The result of a join is a *composite relation* (or table), which I will refer to as the *input relation*. It is the relation that is provided as the input or starting point for all subsequent (filtering) operations in the SELECT command.

It is perhaps easiest to start with an example. The `foods` table has a column `type_id`. As it turns out, the values in this column correspond to values in the `id` column in the `food_types` table. A relationship exists between the two tables. Any value in the `foods.type_id` column must correspond to a value in the `food_types.id` column, and the `id` column is the *primary key* (described later) of `food_types`. The `foods.type_id` column, by virtue of this relationship, is called a *foreign key*: it contains (or references) values in the primary key of another table. This relationship is called a *foreign key relationship*.

Using this relationship, it is possible to join the `foods` and `food_type` tables on these two columns to make a new relation, which provides more detailed information, namely the `food_types.name` for each food in the `foods` table. This is done with the following SQL:

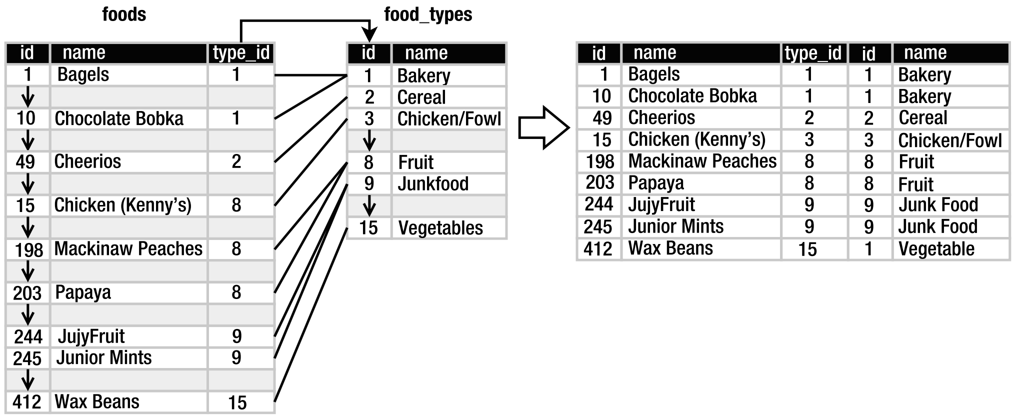
```
sqlite> SELECT foods.name, food_types.name
        FROM foods, food_types
        WHERE foods.type_id=food_types.id LIMIT 10;

name                        name
-------------------------   ---------------
Bagels                      Bakery
Bagels, raisin              Bakery
Bavarian Cream Pie          Bakery
Bear Claws                  Bakery
Black and White cookies     Bakery
Bread (with nuts)           Bakery
Butterfingers               Bakery
Carrot Cake                 Bakery
Chips Ahoy Cookies          Bakery
Chocolate Bobka             Bakery
```

You can see the `foods.name` in the first column of the result, followed by the `food_types.name` in the second. Each row in the former is linked to its associated row in the latter using the `foods.type_id` → `food_types.id` relationship (Figure 4-11).

■ **Note** I am using a new notation in this example to specify the columns in the `SELECT` clause. Rather than specifying just the column names, I am using the notation `table_name.column_name`. The reason is because I have multiple tables in the `SELECT` statement. The database is smart enough to figure out which table a column belongs to—as long as that column name is unique among all tables. If you use a column whose name is also defined in other tables of the join, the database will not be able to figure out which of the columns you are referring to, and will return an error. In practice, when you are joining tables, it is always a good idea to use the `table_name.column_name` notation to avoid any possible ambiguity. This is explained in detail in the section "Names and Aliases."

To carry out the join, the database finds these matched rows. For each row in the first table, the database finds all rows in the second table that have the same value for the joined columns and includes them in the input relation. So in this example, the `FROM` clause built a composite relation by joining the rows of two tables.



foods

id	name	type_id
1	Bagels	1
↓		
10	Chocolate Bobka	1
↓		
49	Cheerios	2
↓		
15	Chicken (Kenny's)	8
↓		
198	Mackinaw Peaches	8
↓		
203	Papaya	8
↓		
244	JujyFruit	9
245	Junior Mints	9
↓		
412	Wax Beans	15

food_types

id	name
1	Bakery
2	Cereal
3	Chicken/Fowl
↓	
8	Fruit
9	Junkfood
↓	
15	Vegetables

id	name	type_id	id	name
1	Bagels	1	1	Bakery
10	Chocolate Bobka	1	1	Bakery
49	Cheerios	2	2	Cereal
15	Chicken (Kenny's)	3	3	Chicken/Fowl
198	Mackinaw Peaches	8	8	Fruit
203	Papaya	8	8	Fruit
244	JujyFruit	9	9	Junk Food
245	Junior Mints	9	9	Junk Food
412	Wax Beans	15	1	Vegetable

Figure 4-11. *foods and food_types join*

The subsequent operations (`WHERE`, `GROUP BY`, etc.) work exactly the same. It is only the input that has changed through joining tables. However, the predicate in the `WHERE` clause (`foods.type_id=food_types.id`) controls what records are returned from the join. This happens in restriction. You might be wondering how this can be if restriction takes place after joining. Well, the short answer is that the database picks up on this by seeing two tables specified in the `FROM` clause. But this still doesn't explain how anything in the `WHERE` clause can have any effect on anything in the `FROM` clause, which is performed before it. You'll find out shortly.

As it turns out, there are six different kinds of joins. The one just described, called an *inner join*, is the most common.

Inner Joins

An inner join is where two tables are joined by a relationship between two columns in the tables, as in this previous example. It is the most common (and perhaps most generally useful) type of join.

An inner join uses another set operation in relational algebra, called an *intersection*. An intersection of two sets produces a set containing elements that exist in both sets. Figure 4-12 illustrates this. The intersection of the set {1, 2, 8, 9} and the set {1, 3, 5, 8} is the set {1, 8}. The intersection operation is represented by a Venn diagram showing the common elements of both sets.

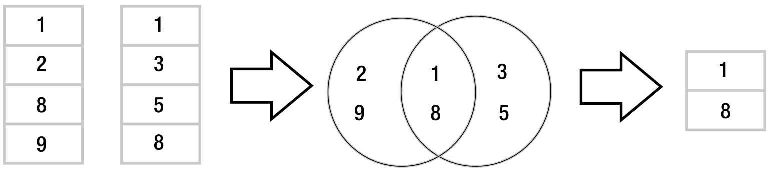
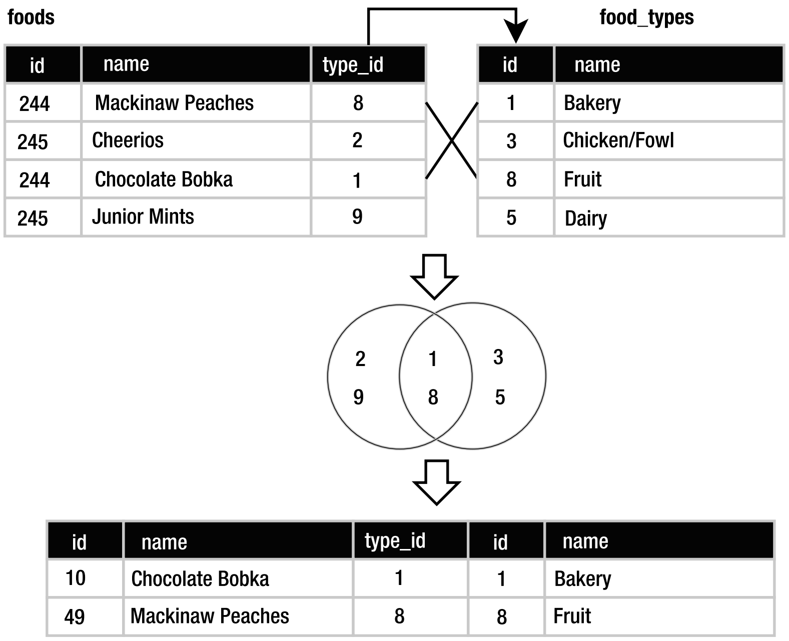


Figure 4-12. *Set intersection*

This is precisely how an inner join works, but the sets in a join are common elements of the related columns. Pretend that the left-hand set in Figure 4-12 represents values in the `foods.type_id` column and the right-hand set represents values of the `food_types.id` column. Given the matching columns, an inner join finds the rows from both sides that contain like values and combines them to form the rows of the result (Figure 4-13). Note that this example assumes that the records shown are the only records in `foods` and `food_types`.



id	name	type_id
244	Mackinaw Peaches	8
245	Cheerios	2
244	Chocolate Bobka	1
245	Junior Mints	9

id	name
1	Bakery
3	Chicken/Fowl
8	Fruit
5	Dairy

id	name	type_id	id	name
10	Chocolate Bobka	1	1	Bakery
49	Mackinaw Peaches	8	8	Fruit

Figure 4-13. *Food join as intersection*

Inner joins only return rows that satisfy the given column relationship, also called the *join condition*. They answer the question, "What rows of *B* match rows in *A* given the following relationship?" Consider the two hypothetical tables shown in Figure 4-14. They both have three columns, with one in common (named `a`). Table `B` is shaded just for illustration purposes.

a	b	c
1	4	7
2	5	8
3	6	9

A

a	d	e
1	4	7
2	5	8
4	6	9

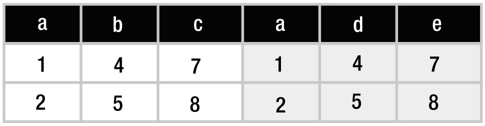
B

Figure 4-14. *Two hypothetical tables*

You can see that two rows in B match two rows in A with respect to column a. This is the inner join:

```
SELECT * FROM A, B where A.a=B.a;
```

The result is shown in Figure 4-15.



a	b	c	a	d	e
1	4	7	1	4	7
2	5	8	2	5	8

Figure 4-15. *Inner join of A and B*

Notice that the third row of B does not match any row in A for this condition (and vice versa). Therefore only two rows are included in the result. The join condition (e.g., A.a=B.a) is what makes this an inner join as opposed to another type of join.

Cross Joins

Imagine for a moment that there were no join condition. What would you get? In this case, if the two tables were not related in any way, SELECT would produce a more fundamental kind of join (the *most* fundamental), which is called a *cross join* or *Cartesian join* (if you read Chapter 3, you will know this as the *cross product* or *Cartesian product*). The Cartesian join is one of the fundamental relational operations. It is a brute-force, almost nonsensical join that results in the combination of all rows from the first table with all rows in the second. A cross join of tables A and B can be expressed with the following pseudocode:

```
for each record a in A
    for each record b in B
        make record a + b
```

In SQL, the cross join of A and B is expressed as follows:

```
SELECT * FROM A,B;
```

FROM, in the absence of anything else, produces a cross join. The result is shown in Figure 4-16. Every row in A is combined with every row in B. In a cross join, no relationship exists between the rows; there is no join condition but they are simply jammed together.

This, then, is what the WHERE clause was filtering with the (inner) join condition in the preceding example. It was removing all those rows in the cross join that had no sensible relationship. An inner join is a subset of a cross join. A cross join contains every possible combination of rows in two tables, whereas an inner join contains only those rows that satisfy a specific relationship between columns in the two tables.

a	b	c	a	d	e
1	4	7	1	4	7
1	4	7	2	5	8
1	4	7	4	6	9
2	5	8	1	4	7
2	5	8	2	5	8
2	5	8	4	6	9
3	6	9	1	4	7
3	6	9	2	5	8
3	6	9	4	6	9

Figure 4-16. *Cross join of tables A and B*

From a purely relational sense, a join is composed of the following set operations, in order:

1. **Cross join**: Take the cross product of all tables in the source list.
2. **Restrict**: Apply the join condition (and any other restrictions) to the cross product to narrow it down. An inner join takes the intersection of related columns to select matching rows. Other joins use other criteria.
3. **Project**: Select the desired columns from the restriction.

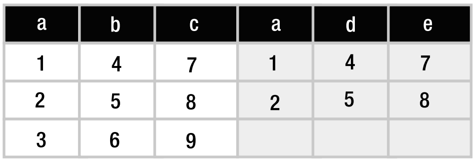
In this sense, all joins then begin as a cross join of all tables in the FROM clause, producing a set of every combination of rows therein. This is the mathematical process. It is most certainly *not* the process used by relational databases. It would be wasteful, to say the least, for databases to blindly start every join by computing the cross product of all tables listed in the FROM clause. There are many ways databases optimize this process so as to avoid combining rows that will simply be thrown out by the WHERE clause. Thus, the mathematical concept and the database implementation are completely different. However, the end results are logically equivalent.

Outer Joins

Three of the remaining four joins are called *outer joins*. An inner join selects rows across tables according to a given relationship. An outer join selects all of the rows of an inner join plus some rows outside of the relationship. The three outer join types are called *left*, *right*, and *full*. A left join operates with respect to the "left table" in the SQL command. For example, in the command

```
SELECT * FROM A LEFT JOIN B ON A.a=B.a;
```

table A is the left table here. The left join favors it. It is the table of significance in a left join. The left join tries to match every row of A with every row in B per the join condition (A.a=B.a). All such matching rows are included in the result. However, the remaining rows of A that don't match B are still included in the result. In this case, these rows have empty values for the B columns. The result is shown in Figure 4-17.



a	b	c	a	d	e
1	4	7	1	4	7
2	5	8	2	5	8
3	6	9			

Figure 4-17. *Left join of A and B*

Remember that the third row of A has no matching row in B with respect to column a. Despite this, the left join includes all rows of A, matching or not.

The right join works similarly, except the right table is the one whose rows are included, matching or not. Operationally, left and right joins are identical; they do the same thing. They differ only in order and syntax. You could argue that there never is any need to do a right join as the only thing it does is swap the arguments in a left join.

A full outer join is the combination of a left and right outer join. It includes all matching records, followed by unmatched records in the right and left tables. Currently, both right and full outer joins are not supported in SQLite. However, as mentioned earlier, a right join can be replaced with a left join, and a full outer join can be performed using compound queries (see the section “Compound Queries” later in this chapter).

Natural Joins

The last join on the list is called a *natural join*. It is actually an inner join in disguise, but with a little syntax and convention thrown in. A natural join joins two tables by their common column names. Thus, using the natural join you can get the inner join of A and B without having to add the join condition A.a=B.a:

```
sqlite> SELECT * FROM A NATURAL JOIN B;

a           b           c           d           e
----------  ----------  ----------  ----------  ----------
1           4           7           4           7
2           5           8           5           8
```

Here, the natural join automatically detects the common column names in A and B (in this case just column a) and links them together. In practice, it is a good idea to avoid natural joins in your applications if you can. Natural joins will join *all* columns by the same name in both tables. Just the process of adding to or removing a column from a table can drastically change the results of a natural join query. Say a program uses a natural join query on A and B. Then suppose someone comes along and adds a new column e to A. That will cause the natural join (and thus the program) to produce completely different results. It’s always better to explicitly define the join conditions of your queries than rely on the semantics of the table schema.

Preferred Syntax

Syntactically, there are various ways of specifying a join. The inner join example of A and B illustrates performing a join implicitly in the WHERE clause:

```
SELECT * FROM A,B WHERE A.a=B.a;
```

When the database sees more than one table listed, it knows there will be a join—at the very least a cross join. The WHERE clause here calls for an inner join.

This implicit form, while rather clean, is actually an older form of syntax that you should avoid. The politically correct way (per SQL92) to express a join in SQL is using the JOIN keyword. The general form is

```
SELECT heading FROM LEFT_TABLE join_type RIGHT_TABLE ON join_condition;
```

This explicit form can be used for all join types. For example:

```
SELECT * FROM A INNER JOIN B ON A.a=B.a;
SELECT * FROM A LEFT JOIN B ON A.a=B.a;
SELECT * FROM A NATURAL JOIN B ON A;
SELECT * FROM A CROSS JOIN B ON A;
```

Finally, when the join condition is based on columns that share the same name, it can be simplified with the USING keyword. USING simply names the common column (or columns) to include in the join condition:

```
SELECT * FROM A INNER JOIN B USING(a);
```

The argument of USING is a comma-separated list of column names within parentheses.

Joins are processed from left to right. Consider the mutliway join shown in Figure 4-18.

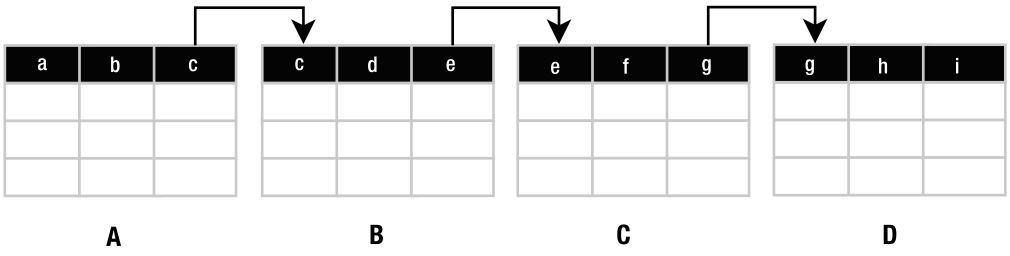


Figure 4-18. *Multiway join*

Using the preferred form, you simply keep tacking on additional tables by adding more join expressions to the end of the statement:

```
SELECT * FROM A JOIN B USING (c) JOIN C USING (e) JOIN D USING (g);
```

The first relation, *R1*, is from A join B on c. The second relation, *R2*, is R1 join C on e. The final relation, *R3*, is R2 join D on g. This is shown in Figure 4-19.

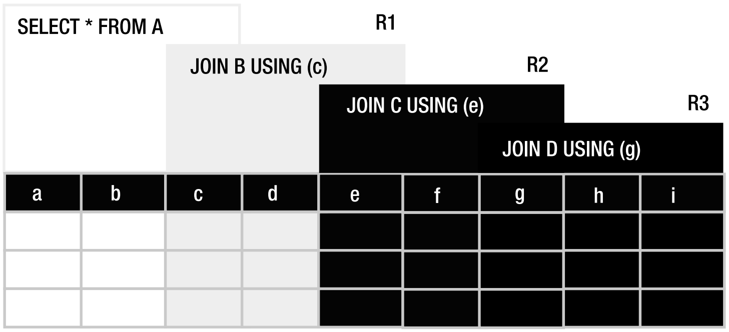


Figure 4-19. *A politically correct multiway join construction*

Names and Aliases

When joining tables, column names can become ambiguous. For example, A and B as defined in Figure 4-14 both have a column a. What if they have another column named name, and you just want to select the name column of B but not A? To help with this type of task, you can qualify column names with their table names to remove any ambiguity. So in this example, you could write

```
SELECT B.name FROM A JOIN B USING (a);
```

Another useful feature is *aliases*. If your table name is particularly long, and you don't want to have to use its name every time you qualify a column, you can use an alias. Aliasing is actually a fundamental relational operation called *rename*. The rename operation simply assigns a new name to a relation. For example, consider the statement:

```
SELECT foods.name, food_types.name FROM foods, food_types
WHERE foods.type_id=food_types.id LIMIT 10;
```

There is a good bit of typing here. You can rename the tables in the source clause by simply including the new name directly after the table name, as in the following example:

```
SELECT f.name, t.name FROM foods f, food_types t
WHERE f.type_id=t.id LIMIT 10;
```

Here, the foods table is assigned the alias f, and the food_types table is assigned the alias t. Now, every other reference to foods or food_types in the statement must use the alias f and t, respectively. Aliases make it possible to do *self-joins*—joining a table with itself. For example, say you want to know what foods in season 4 are mentioned in other seasons. You would first need to get a list of episodes and foods in season 4, which you would obtain by joining episodes and episodes_foods. But then you would need a similar list for foods outside of season 4. Finally, you would combine the two lists based on their common foods. The following query uses self-joins to do the trick:

```
SELECT f.name as food, e1.name, e1.season, e2.name, e2.season
FROM episodes e1, foods_episodes fe1, foods f,
     episodes e2, foods_episodes fe2
```

```
WHERE
  -- Get foods in season 4
  (e1.id = fe1.episode_id AND e1.season = 4) AND fe1.food_id = f.id
  -- Link foods with all other epsisodes
  AND (fe1.food_id = fe2.food_id)
  -- Link with their respective episodes and filter out e1's season
  AND (fe2.episode_id = e2.id AND e2.season != e1.season)
ORDER BY f.name;

food                  name               season  name                     season
--------------------  -----------------  ------  -----------------------  ------
Bouillabaisse         The Shoes          4       The Stake Out            1
Decaf Cappuccino      The Pitch          4       The Good Samaritan       3
Decaf Cappuccino      The Ticket         4       The Good Samaritan       3
Egg Salad             The Trip 1         4       Male Unbonding           1
Egg Salad             The Trip 1         4       The Stock Tip            1
Mints                 The Trip 1         4       The Cartoon              9
Snapple               The Virgin         4       The Abstinence           8
Tic Tacs              The Trip 1         4       The Merv Griffin Show    9
Tic Tacs              The Contest        4       The Merv Griffin Show    9
Tuna                  The Trip 1         4       The Stall                5
Turkey Club           The Bubble Boy     4       The Soup                 6
Turkey Club           The Bubble Boy     4       The Wizard               9
```

I have put comments in the SQL to better explain what is going on. This example uses two self-joins. There are two instances of `episodes` and `foods_episodes`, but they are treated as if they are two independent tables. The query joins `foods_episodes` back on itself to link the two instances of `episodes`. Furthermore, the two `episodes` instances are related to each other by an inequality condition to ensure that they are in different seasons.

You can alias column names and expressions in the same way. For example, to get the top ten episodes with the most foods, nicely labeled, you'd use this:

```
sqlite> SELECT e.name AS Episode, COUNT(f.id) AS Foods
        FROM foods f
          JOIN foods_episodes fe on f.id=fe.food_id
          JOIN episodes e on fe.episode_id=e.id
        GROUP BY e.id
        ORDER BY Foods DESC
        LIMIT 10;

Episode               Foods
--------------------  -----
The Soup              23
The Fatigues          14
The Bubble Boy        12
The Finale 1          10
The Merv Griffin Show 9
The Soup Nazi         9
The Wink              9
The Dinner Party      9
The Glasses           9
The Mango             9
```

Note that the AS keyword is optional. I just use it because it seems more legible that way to me. Column aliases will change the column headers returned in the result set. You may refer to columns or expressions by their aliases elsewhere in the statement if you wish (as in the ORDER BY clause in the preceding example), but you are not required to do so like you are with tables.

Subqueries

Subqueries are SELECTs within SELECTs. They are also called *subselects*. Subqueries are useful in many ways, they work anywhere normal expressions work, and they can therefore be used in a variety of places in a SELECT statement. Subqueries are useful in other commands as well.

Perhaps the most common use of subqueries is in the WHERE clause, specifically using the IN operator. The IN operator is a binary operator that takes an input value and a list of values and returns true if the input value exists in the list, or false otherwise. Here's an example:

```
sqlite> SELECT 1 IN (1,2,3);
1
sqlite> SELECT 2 IN (3,4,5);
0
sqlite> SELECT COUNT(*) FROM foods WHERE type_id IN (1,2);
62
```

Using a subquery, you can rewrite the last statement in terms of names from the food_types:

```
sqlite> SELECT COUNT(*) FROM foods WHERE type_id
        IN (SELECT id FROM food_types WHERE name='Bakery' OR name='Cereal');

62
```

Subqueries in the SELECT clause can be used to add additional data from other tables to the result set. For example, to get the number of episodes each food appears in, the actual count from foods_episodes can be performed in a subquery in the SELECT clause:

```
sqlite> SELECT name,
        (SELECT COUNT(id) FROM foods_episodes WHERE food_id=f.id) count
        FROM foods f ORDER BY count DESC LIMIT 10;

name            count
----------      ----------
Hot Dog         5
Pizza           4
Ketchup         4
Kasha           4
Shrimp          3
Lobster         3
Turkey Sandwich 3
Turkey Club     3
Egg Salad       3
Tic Tacs        3
```

The ORDER BY and LIMIT clauses here serve to create a top ten list, with hot dogs at the top. Notice that the subquery's predicate references a table in the enclosing SELECT command: food_id=f.id. The variable f.id exists in the outer query. The subquery in this example is called a *correlated subquery* because it references, or correlates to, a variable in the outer (enclosing) query.

Subqueries in the SELECT clause are also helpful with computing percentages of aggregates. For example, the following SQL breaks foods down by types and their respective percentages:

```
SELECT (SELECT name FROM food_types WHERE id=f.type_id) Type,
COUNT(type_id) Items,
COUNT(type_id)*100.0/(SELECT COUNT(*) FROM foods) as Percentage
FROM foods f GROUP BY type_id ORDER BY Percentage DESC;
```

```
Type            Items    Percentage
--------------  -------  --------------
Junkfood        61       14.76997578692
Drinks          60       14.52784503631
Bakery          48       11.62227602905
Meat            36       8.716707021791
Vegetables      32       7.748184019370
Chicken/Fowl    23       5.569007263922
Fruit           23       5.569007263922
Sandwiches      23       5.569007263922
Condiments      22       5.326876513317
Soup            19       4.600484261501
Dairy           17       4.116222760290
Rice/Pasta      16       3.874092009685
Cereal          15       3.631961259079
Seafood         14       3.389830508474
Dip             4        0.968523002421
```

Here, a subquery must be used as the divisor rather than COUNT() because the statement uses a GROUP BY. Remember that aggregates in GROUP BY are applied to groups, not the entire result set; therefore the subquery's COUNT() must be used to get the total rows in foods.

Subqueries can be used in the ORDER BY clause as well. The following SQL groups foods by the size of their respective food groups, from greatest to least:

```
SELECT * FROM foods f
ORDER BY (SELECT COUNT(type_id)
FROM foods WHERE type_id=f.type_id) DESC;
```

ORDER BY in this case does not refer to any specific column in the result. How does this work then? The ORDER BY subquery is run for each row, and the result is associated with the given row. You can think of it as an invisible column in the result set, which is used to order rows.

Finally, we have the FROM clause. There may be times when you want to join only part of a table rather than all of it. Yet another job for a subquery:

```
SELECT f.name, types.name FROM foods f
INNER JOIN (SELECT * FROM food_types WHERE id=6) types
ON f.type_id=types.id;
```

```
name                       name
-------------------------  -----
Generic (as a meal)        Dip
Good Dip                   Dip
Guacamole Dip              Dip
Hummus                     Dip
```

Notice that the use of a subquery in the FROM clause requires a rename operation. In this case, the subquery was named types. This query could have been written using a full join of food_types, of course, but it may have incurred more overhead as there would have been more records to match.

Another use of subqueries is in reducing the number of rows in an aggregating join. Consider the following query:

```
SELECT e.name name, COUNT(fe.food_id) foods FROM episodes e
INNER JOIN foods_episodes fe ON e.id=fe.episode_id
GROUP BY e.id
ORDER BY foods DESC
LIMIT 10;
```

```
name            foods
--------------  -------
The Soup        23
The Fatigues    14
The Bubble Boy  12
The Finale 1    10
The Mango       9
The Glasses     9
The Dinner Par  9
The Wink        9
The Soup Nazi   9
The Merv Griff  9
```

This query lists the top ten shows with the most food references. The join here must match 181 rows in episodes with 502 rows in foods_episodes (181/502). Then it has to compute the aggregate. The fewer rows it has to match, the less work it has to do and the more efficient (and faster) the query becomes. Aggregation collapses rows, right? Then a great way to reduce the number of rows the join must match is to aggregate *before* the join. The only way to do this is with a subquery:

```
SELECT e.name, agg.foods FROM episodes e
INNER JOIN
(SELECT fe.episode_id as eid, count(food_id) as foods
   FROM foods_episodes fe
   GROUP BY episode_id ) agg
ON e.id=agg.eid
ORDER BY agg.foods DESC
LIMIT 10;
```

This query moves aggregation into a subquery that is run before the join, the results of which are joined to the `episodes` table for the final result. The subquery produces one aggregate (the number of foods) per episode. There are 181 episodes. Thus, this query reduces the join size from 181:502 to 181:181. However, since we are only looking for the top ten food-referencing episodes, we can also move the `LIMIT 10` clause into the subquery as well and reduce the number still, to 181:10:

```
SELECT e.name, agg.foods FROM episodes e
INNER JOIN
(SELECT fe.episode_id as eid, count(food_id) as foods
   FROM foods_episodes fe
   GROUP BY episode_id
   ORDER BY foods DESC LIMIT 10) agg
ON e.id=agg.eid
ORDER BY agg.foods DESC;
```

This subquery returns only ten rows, which correspond to the top ten food shows. The join disrupts the descending order of the subquery so another `ORDER BY` is required in the main query to reestablish it.

The thing to remember about subqueries is that they can be used *anywhere* a relational expression can be used. A good way to learn how, where, and when to use them is to just play around with them and see what you can get away with. There is often more than one way to skin a cat in SQL. When you understand the big picture, you can make more informed decisions on when a query might be rewritten to run more efficiently.

Compound Queries

Compound queries are kind of the inverse of subqueries. A compound query is a query that processes the results of multiple queries using three specific relational operations: union, intersection, and difference. In SQL, these are defined using the `UNION`, `INTERSECT`, and `EXCEPT` keywords, respectively.

Compound query operations require a few things of their arguments:

- The relations involved must have the same number of columns (degree).
- There can only be one `ORDER BY` clause, which is at the end of the compound query, and applies to the combined result.

Furthermore, relations in compound queries are processed from left to right.

The INTERSECT operation takes two relations *A* and *B*, and selects all rows in *A* that also exist in *B*. The following SQL uses INTERSECT to find the all-time top ten foods that appear in seasons 3 through 5:

```
SELECT f.* FROM foods f
INNER JOIN
  (SELECT food_id, count(food_id) as count FROM foods_episodes
     GROUP BY food_id
     ORDER BY count(food_id) DESC LIMIT 10) top_foods
  ON f.id=top_foods.food_id
INTERSECT
SELECT f.* FROM foods f
  INNER JOIN foods_episodes fe ON f.id = fe.food_id
  INNER JOIN episodes e ON fe.episode_id = e.id
  WHERE e.season BETWEEN 3 and 5
ORDER BY f.name;
```

```
id     type_id  name
-----  -------  --------------------
4      1        Bear Claws
146    7        Decaf Cappuccino
153    7        Hennigen's
55     2        Kasha
94     4        Ketchup
164    7        Naya Water
317    11       Pizza
```

To produce the top ten foods, I needed an ORDER BY in the first SELECT statement. Since compound queries only allow one ORDER BY at the end of the statement, I got around this by performing an inner join on a subquery in which I computed the top ten most common foods. Subqueries can have ORDER BY clauses because they run independently of the compound query. The inner join then produces a relation containing the top ten foods. The second query returns a relation containing all foods in episodes 3 through 5. The INTERSECT operation then finds all matching rows.

The EXCEPT operation takes two relations *A* and *B* and finds all rows in *A* that are not in *B*. By changing the INTERSECT to EXCEPT in the previous example, you can find which top ten foods are *not* in seasons 3 through 5:

```
SELECT f.* FROM foods f
INNER JOIN
  (SELECT food_id, count(food_id) AS count FROM foods_episodes
     GROUP BY food_id
     ORDER BY count(food_id) DESC LIMIT 10) top_foods
  ON f.id=top_foods.food_id
```

```
EXCEPT
SELECT f.* FROM foods f
  INNER JOIN foods_episodes fe ON f.id = fe.food_id
  INNER JOIN episodes e ON fe.episode_id = e.id
  WHERE e.season BETWEEN 3 and 5
ORDER BY f.name;
```

```
id     type_id  name
-----  -------  --------
192    8        Banana
133    7        Bosco
288    10       Hot Dog
```

As mentioned earlier, what is called the EXCEPT operation in SQL is referred to as the *difference* operation in relational algebra.

The UNION operation takes two relations, *A* and *B*, and combines them into a single relation containing all distinct rows of *A* and *B*. In SQL, UNION combines the results of two SELECT statements. By default, UNION eliminates duplicates. If you want duplicates included in the result, then use UNION ALL. For example, the following SQL finds the single most and single least frequently mentioned foods:

```
SELECT f.*, top_foods.count FROM foods f
INNER JOIN
  (SELECT food_id, count(food_id) AS count FROM foods_episodes
     GROUP BY food_id
     ORDER BY count(food_id) DESC LIMIT 1) top_foods
  ON f.id=top_foods.food_id
UNION
SELECT f.*, bottom_foods.count FROM foods f
INNER JOIN
  (SELECT food_id, count(food_id) AS count FROM foods_episodes
     GROUP BY food_id
     ORDER BY count(food_id) LIMIT 1) bottom_foods
  ON f.id=bottom_foods.food_id
ORDER BY top_foods.count DESC;
```

```
id     type_id  name       top_foods.count
-----  -------  ---------  ---------------
288    10       Hot Dog    5
1      1        Bagels     1
```

Both queries return only one row. The only difference in the two is which way they sort their results. The UNION simply combines the two rows into a single relation.

UNION ALL also provides a way to implement a full outer join. You simply perform a left join in the first SELECT, a right join (written in the form of a left join) in the second, and the result is a full outer join.

Compound queries are useful when you need to process similar data sets that are materialized in different ways. Basically, if you cannot express everything you want in a single SELECT statement, you can use a compound query to get part of what you want in one SELECT statement and part in another (and perhaps more), and process the sets accordingly. For example, the INTERSECT example earlier compared two lists of foods that could not be materialized with a single SELECT statement. One list contained the top ten foods, and the second contained all foods that appeared in specific seasons. And I wanted the top ten foods that appeared in the specific seasons. A compound query was the only way to get this information. The same was true in the UNION example. It is not possible to select just the first and last records of a result set. The UNION took two slight variations of a query to get the first and last records and combined the results. Also, as with joins, you can string as many SELECT commands together as you like by using any of these compound operations.

Conditional Results

The CASE expression allows you to handle various conditions within a SELECT statement. There are two forms. The first and simplest form takes a static value and lists various case values linked to return values:

```
CASE value
  WHEN x THEN value_x
  WHEN y THEN value_y
  WHEN z THEN value_z
  ELSE default_value
END
```

Here's a simple example:

```
SELECT name || CASE type_id
                 WHEN 7  THEN ' is a drink'
                 WHEN 8  THEN ' is a fruit'
                 WHEN 9  THEN ' is junkfood'
                 WHEN 13 THEN ' is seafood'
                 ELSE NULL
               END description
FROM foods
WHERE description IS NOT NULL
ORDER BY name
LIMIT 10;
```

```
description
---------------------------------------------
All Day Sucker is junkfood
Almond Joy is junkfood
Apple is a fruit
Apple Cider is a drink
Apple Pie is a fruit
Arabian Mocha Java (beans) is a drink
Avocado is a fruit
Banana is a fruit
Beaujolais is a drink
Beer is a drink
```

The CASE expression in this example handles a few different type_id values, returning a string appropriate for each one. The returned value is called description, as qualified after the END keyword. This string is concatenated to name by the string concatenation operator (||), making a complete sentence. For all type_ids not specified in a WHEN condition, CASE returns NULL. The SELECT statement filters out such NULL values in the WHERE clause, so all that is returned are rows that the CASE expression does handle.

The second form of CASE allows for expressions in the WHEN condition. It has the following form:

```
CASE
  WHEN condition1 THEN value1
  WHEN condition2 THEN value2
  WHEN condition3 THEN value3
  ELSE default_value
END
```

CASE works equally well in subselects comparing aggregates. The following SQL picks out frequently mentioned foods:

```
SELECT name,(SELECT
               CASE
                 WHEN count(*) > 4
                   THEN 'Very High'
                 WHEN count(*) = 4
                   THEN 'High'
                 WHEN count(*) IN (2,3)
                   THEN 'Moderate'
                 ELSE 'Low'
               END
             FROM foods_episodes
             WHERE food_id=f.id) frequency
FROM foods f
WHERE frequency LIKE '%High'
```

```
name       frequency
---------  ----------
Kasha      High
Ketchup    High
Hot Dog    Very High
Pizza      High
```

This query runs a subquery for each row in `foods` that classifies the food by the number of episodes it appears in. The result of this subquery is included as a column called `frequency`. The `WHERE` predicate filters `frequency` values that have the word "High" in them.

Only one condition is executed in a `CASE` expression. If more than one condition is satisfied, only the first of them is executed. If no conditions are satisfied and no `ELSE` condition is defined, `CASE` returns `NULL`.

The Thing Called Null

Most relational databases support a special value called `NULL`. `NULL` is a placeholder for missing information. `NULL` is not a value per se. Rather, `NULL` is the absence of a value. Better yet, it is a value denoting the absence of a value. Some say it stands for "unknown," "not applicable," or "not known." But truth be told, `NULL` is still rather vague and mysterious. Try as you may to nail it down, `NULL` can still play with your mind: `NULL` is not nothing; `NULL` is not something; `NULL` is not true; `NULL` is not false; `NULL` is not zero. Simply put, `NULL` is resolutely what it is: `NULL`. And not everyone can agree on what that means. To some, `NULL` is a four-letter word. `NULL` rides a Harley, sports racy tattoos, and refuses to conform. To others, it is a necessary evil and serves an important role in society. You may love it. You may hate it. But it's here to stay. And if you are going to keep company with `NULL`, you'd better know what you are getting yourself into.

Based on what you already know, it should come as no surprise to learn that even the SQL standard is not completely clear on how to deal with `NULL` in all cases. Regardless, the SQLite community came to a consensus by evaluating how a number of other major relational databases handled `NULL` in particular ways. The result was that there was some but not total consistency in how they all worked. Oracle, PostgreSQL, and DB2 were almost identical with respect to `NULL` handling, so SQLite's approach was to be compatible with them.

Working with `NULL` is appreciably different than working with any other kind of value. For example, if you are looking for rows in `foods` whose name is `NULL`, the following SQL is useless:

```
SELECT * FROM foods WHERE foods.name=NULL;
```

It won't return any rows, period. The problem here is that the expression *anything*`=NULL` evaluates to `NULL` (even the expression `NULL=NULL` is `NULL`). And `NULL` is not true (nor is it false), so the `WHERE` clause will never evaluate to true, and therefore no rows will be selected by the query in its current form. In order to get it to work as intended, you must recast the query to use the `IS` operator:

```
SELECT * FROM foods WHERE foods.name IS NULL;
```

The `IS` operator properly checks for a `NULL` and returns true if it finds one. If you want values that are not `NULL`, then use `IS NOT NULL`.

But this is just the beginning of our `NULL` fun. `NULL` has a kind of Midas-like quality in that everything it touches turns to `NULL`. For example:

```
sqlite> SELECT NULL=NULL;
NULL
sqlite> SELECT NULL OR NULL;
NULL
sqlite> SELECT NULL AND NULL;
NULL
sqlite> SELECT NOT NULL;
NULL
sqlite> SELECT 9E9 - 1E-9*NULL;
NULL
```

Additionally, it is important to note that `COUNT(*)` and `COUNT(column)` are distinctly different with respect to how they handle `NULL`. `COUNT(*)` counts rows, regardless of any particular column value, so `NULL` has no effect on it. `COUNT(column)`, on the other hand, only counts the rows with non-`NULL` values in `column`, and rows where `column` is `NULL` are ignored.

In order to accommodate `NULL` in logical expressions, SQL uses something called *three-value* (or *tristate*) logic, where `NULL` is one of the truth values. The truth table for logical `AND` and logical `OR` with `NULL` thrown into the mix is shown in Table 4-5.

Table 4-5. *AND and OR with NULL*

x	y	x AND y	y OR y
True	True	True	True
True	False	False	True
True	NULL	NULL	True
False	False	False	False
False	NULL	False	NULL
NULL	NULL	NULL	NULL

Since a single `NULL` can nullify an entire expression, it seems that there should be some way to specify a suitable default in case a `NULL` crops up. To that end, SQLite provides a function for dealing with `NULL`, called `COALESCE`, which is part of the SQL99 standard. `COALESCE` takes a list of values and returns the first non-`NULL` in the list. Take the following example:

```
SELECT item.price-(SELECT SUM(amount) FROM discounts WHERE id=item.id)
FROM products WHERE …
```

This is a hypothetical example of a query that needs to calculate the price of an item with any discounts applied. But what if the subquery returns `NULL`? Then the whole calculation and therefore the price becomes `NULL`. That’s not good. `COALESCE` provides a barrier through which `NULL` may not pass:

```
SELECT item.price-(SELECT COALESCE(SUM(amount),0)
                   FROM discounts WHERE id=item.id)
FROM products WHERE …
```

In this case if `SUM(amount)` turns out to be `NULL`, `COALESCE` will return 0 instead, taking the teeth out of `NULL`.

Conversely, the `NULLIF` function takes two arguments and returns `NULL` if they have the same values; otherwise, it returns the first argument:

```
sqlite> SELECT NULLIF(1,1);
NULL
sqlite> SELECT NULLIF(1,2);
1
```

If you use `NULL`, you need to take special care in queries that refer to columns that may contain `NULL` in their predicates and aggregates. `NULL` can do quite a number on aggregates if you are not careful. Consider the following example:

```
sqlite> CREATE TABLE sales (product_id int, amount real, discount real);
sqlite> INSERT INTO sales VALUES (1, 10.00, 1.00);
sqlite> INSERT INTO sales VALUES (2, 10000.00, NULL);
sqlite> SELECT * FROM sales;

product_id  amount      discount
----------  ----------  ----------
1           10          1
2           10000       NULL
```

You have a sales table that contains the products sold throughout the day. It holds the product ID, the price, and the total discounts that were included in the sale. There are two sales in the table: one is a $10 purchase with a $1 discount. Another is a $10,000 purchase with no discount, represented by a `NULL`, as in "not applicable." At the end of the day you want to tabulate net sales after discounts. You try the following:

```
sqlite> SELECT SUM(amount-discount) FROM sales;

SUM(amount-discount)
--------------------
9.0
```

Where did the $10,000 sale go? Well, 10,000 minus `NULL` is `NULL`. `NULL`s are weeded out by the `WHERE` clause, and therefore contribute nothing to an aggregate, so it simply disappears. Your calculation is off by 99.9 percent. So, knowing better, you investigate and specifically look for the missing sale using the following SQL:

```
sqlite> SELECT SUM(amount) from sales WHERE amount-discount > 100.00;

NULL
```

What's the problem here? Well, when the database evaluates the `WHERE` clause for the record of interest, the expression becomes 10,000 – `NULL` > 100.00. Breaking this down, the predicate evaluates to `NULL`:

```
(10000 - NULL > 100.00)  →  (NULL > 100.00)  →  NULL
```

Again, NULLs don't pass through WHERE, so the $10,000 row seems invisible.

If you are going to stay in the black, you will need to handle NULL better. If NULLs will be allowed in the discount column, then the queries that use that column have to be NULL-aware:

```
sqlite> SELECT SUM(amount-COALESCE(discount,0)) FROM sales;
10009

sqlite> SELECT SUM(amount) from sales
        WHERE amount-COALESCE(discount,0) > 100.00;
10000.0
```

So, NULL can be useful, and can indeed have a very specific meaning, but using it without understanding the full implications can lead to unpleasant surprises.

Set Operations

Congratulations, you have learned the SELECT command. Not only have you learned how the command works, but you've covered a large part of relational algebra in the process. SELECT contains 12 out of 14 operations defined in relational algebra. Here is a list of all of these operations, along with the parts of SELECT that employ them:

- **Restriction**: Restriction takes a single relation and produces a row-wise subset of it. Restriction is performed by the WHERE, HAVING, DISTINCT, and LIMIT clauses.
- **Projection**: Projection produces a column-wise subset of its input relation. Projection is performed by the SELECT clause.
- **Cartesian product**: The Cartesian product takes two relations, *A* and *B*, and produces a relation whose rows are the composite of the two relations by combining every row of *A* with every other row in *B*. SELECT performs the Cartesian product when multiple tables are listed in the FROM clause and no join condition is provided.
- **Union**: The union operation takes two relations, *A* and *B*, and creates a new relation containing all distinct rows from both *A* and *B*. Both *A* and *B* must have the same degree (number of columns). SELECT performs union operations in compound queries, which employ the UNION keyword.
- **Difference**: The difference operation takes two relations, *A* and *B*, and produces a relation whose rows consist of the rows in *A* that are not in *B*. SELECT performs difference in compound queries that employ the EXCEPT operator.
- **Rename**: The rename operation takes a single relation and assigns it a new name. Rename is used in the FROM clause.
- **Intersection**: The intersection operation takes two relations, *A* and *B*, and produces a relation whose rows are contained in both *A* and *B*. SELECT performs the intersection operation in compound queries that employ the INTERSECT operator.
- **Natural join**: A natural join takes two relations and performs an inner join on them by equating the commonly named columns as the join condition. SELECT performs a natural join with joins that use the NATURAL JOIN clause.

- **Generalized projection**: The generalized projection operation is an extension of the projection operation, which allows the use of arithmetic expressions, functions, and aggregates in the projection list. Generalized projection is used in the `SELECT` clause. Associated with aggregates is the concept of grouping, whereby rows with similar column values can be separated into individual groups. This is expressed in the `SELECT` command using the `GROUP BY` clause. Furthermore, groups can be filtered using the `HAVING` clause, which consists of a predicate similar to that of the `WHERE` clause. The predicate in `HAVING` is expressed in terms of aggregate values.
- **Left outer join**: The left outer join operation takes two relations, *A* and *B*, and returns the inner join of *A* and *B* along with the unmatched rows of *A*. *A* is the first relation defined in the `FROM` clause, and is hence the left relation. The left join includes the unmatched rows of the left relation along with the matched columns in the result.
- **Right and full outer join**: The right outer join operation is similar to the left join, only it includes the unmatched rows of the right relation. The full outer join includes unmatched rows from both relations.

Modifying Data

Compared to the `SELECT` command, the statements used to modify data are quite easy to use and understand. There are three DML statements for modifying data—`INSERT`, `UPDATE`, and `DELETE`—and they do pretty much what their names imply.

Inserting Records

You insert records into a table using the `INSERT` command. `INSERT` works on a single table, and can both insert one row at a time or many rows at once using a `SELECT` command. The general form of the `INSERT` command is as follows:

```
INSERT INTO table (column_list) VALUES (value_list);
```

The variable `table` specifies which table—the target table—to insert into. The variable `column_list` is a comma-separated list of column names, all of which must exist in the target table. The variable `value_list` is a comma-separated list of values that correspond to the names given in `column_list`. The order of values in `value_list` must correspond to the order of columns in `column_list`. For example, you'd use this to insert a row into `foods`:

```
sqlite> INSERT INTO foods (name, type_id) VALUES ('Cinnamon Bobka', 1);
```

This statement inserts one row, specifying two column values. `'Cinnamon Bobka'`—the first value in the value list—corresponds to the column `name`—the first column in the column list. Similarly, the value 1 corresponds to `type_id`, which is listed second. Notice that `id` was not mentioned. In this case, the database uses the default value. Since `id` is declared as `INTEGER PRIMARY KEY`, it will be automatically generated and associated with the record (as explained in the section "Primary Key Constraints"). The inserted record can be verified with a simple `SELECT`:

```
sqlite> SELECT * FROM foods WHERE name='Cinnamon Bobka';

id          type_id     name
----------  ----------  --------------
413         1           Cinnamon Bobka

sqlite> SELECT MAX(id) from foods;

MAX(id)
----------
413

sqlite> SELECT last_insert_rowid();

last_insert_rowid()
-------------------
413
```

Notice that the value 413 was automatically generated for `id`, which is the largest value in the column. Thus, SQLite provided a monotonically increasing value. You can confirm this with the built-in SQL function `last_insert_rowid()`, which returns the last automatically generated key value, as shown in the example.

If you provide a value for every column of a table in `INSERT`, then the column list can be omitted. In this case, the database assumes that the order of values provided in the value list correspond to the order of columns as declared in the `CREATE TABLE` statement. For example:

```
sqlite> INSERT INTO foods VALUES(NULL, 1, 'Blueberry Bobka');
sqlite> SELECT * FROM foods WHERE name LIKE '%Bobka';

id          type_id     name
----------  ----------  ---------------
10          1           Chocolate Bobka
413         1           Cinnamon Bobka
414         1           Blueberry Bobka
```

Notice here the order of arguments. `'Blueberry Bobka'` came after 1 in the value list. This is because of the way the table was declared. To view the table's schema, type `.schema foods` at the shell prompt:

```
sqlite> .schema foods
CREATE TABLE foods(
  id integer primary key,
  type_id integer,
  name text );
CREATE INDEX foods_name_idx on foods (name COLLATE NOCASE);
```

The first column is `id`, followed by `type_id`, followed by `name`. This, therefore, is the order you must list values in `INSERT` statements on `foods`. Why did I use a `NULL` value for `id` in the preceding `INSERT` statement? Because SQLite knows that `id` in `foods` is an autoincrement column, and specifying a NULL is the equivalent of not providing a value at all. This triggers the automatic key generation. It's just a convenient trick. There is no deeper meaning or theoretical basis behind it. We will look at the subtleties of autoincrement columns later in this chapter.

Subqueries can be used in INSERT statements, both as components of the value list and as a complete replacement of the value list. Here's an example:

```
INSERT INTO foods
VALUES (NULL,
       (SELECT id FROM food_types WHERE name='Bakery'),
       'Blackberry Bobka');
SELECT * FROM foods WHERE name LIKE '%Bobka';
```

```
id          type_id     name
----------  ----------  ----------------
10          1           Chocolate Bobka
413         1           Cinnamon Bobka
414         1           Blueberry Bobka
415         1           Blackberry Bobka
```

Here, rather than hard-coding the type_id value, I had SQLite look it up for me. Here's another example:

```
INSERT INTO foods
SELECT last_insert_rowid()+1, type_id, name FROM foods
WHERE name='Chocolate Bobka';
SELECT * FROM foods WHERE name LIKE '%Bobka';
```

```
id          type_id     name
----------  ----------  ----------------
10          1           Chocolate Bobka
413         1           Cinnamon Bobka
414         1           Blueberry Bobka
415         1           Blackberry Bobks
416         1           Chocolate Bobka
```

This query completely replaces the value list with a SELECT statement. As long as the number of columns in the SELECT clause matches the number of columns in the table (or the number of columns in the columns list, if provided), INSERT will work just fine. Here, I added another chocolate bobka and used the expression last_insert_rowid()+1 as the id value. I could have just as easily used NULL instead. In fact, I probably should have used NULL rather than last_insert_rowid(), as last_insert_rowid() will return 0 if you have not previously inserted a row in the current session. I could safely assume that this would work properly for these examples, but it would not be a good idea to make this assumption in a program.

There is nothing stopping you from inserting multiple rows at a time using the SELECT form of INSERT. As long as the number of columns matches, INSERT will insert every row in the result. For example:

```
sqlite> CREATE TABLE foods2 (id int, type_id int, name text);
sqlite> INSERT INTO foods2 SELECT * FROM foods;
sqlite> SELECT COUNT(*) FROM foods2;

COUNT(*)
--------------------
418
```

This creates a new table `foods2` and inserts into it all of the records from `foods`.

However, there is an easier way to do this. The `CREATE TABLE` statement has a special syntax for creating tables from `SELECT` statements. The previous example could have been performed in one step using this syntax:

```
sqlite> CREATE TABLE foods2 AS SELECT * from foods;
sqlite> SELECT COUNT(*) FROM list;

COUNT(*)
--------
418
```

`CREATE TABLE` does both steps in one fell swoop. This can be especially useful for creating temporary tables:

```
CREATE TEMP TABLE list AS
SELECT f.name Food, t.name Name,
       (SELECT COUNT(episode_id)
        FROM foods_episodes WHERE food_id=f.id) Episodes
FROM foods f, food_types t
WHERE f.type_id=t.id;
SELECT * FROM list;
```

```
Food                  Name        Episodes
--------------------  ----------  ----------
Bagels                Bakery      1
Bagels, raisin        Bakery      2
Bavarian Cream Pie    Bakery      1
Bear Claws            Bakery      3
Black and White cook  Bakery      2
Bread (with nuts)     Bakery      1
Butterfingers         Bakery      1
Carrot Cake           Bakery      1
Chips Ahoy Cookies    Bakery      1
Chocolate Bobka       Bakery      1
```

When using this form of `CREATE TABLE`, be aware that any constraints defined in the source table are not created in the new table. Specifically, the autoincrement columns will not be created in the new table, nor will indexes, `UNIQUE` constraints, and so forth.

It is also worth mentioning here that you have to be aware of `UNIQUE` constraints when inserting rows. If you add duplicate values on columns that are declared as `UNIQUE`, SQLite will stop you in your tracks:

```
sqlite> SELECT MAX(id) from foods;

MAX(id)
-------
416

sqlite> INSERT INTO foods VALUES (416, 1, 'Chocolate Bobka');
SQL error: PRIMARY KEY must be unique
```

Updating Records

You update records in a table using the `UPDATE` command. The `UPDATE` command modifies one or more columns within one or more rows in a table. `UPDATE` has the general form

```
UPDATE table SET update_list WHERE predicate;
```

The `update_list` is a list of one or more column assignments of the form `column_name=value`. The `WHERE` clause works exactly as in `SELECT`. Half of `UPDATE` is really a `SELECT` statement. The `WHERE` clause identifies rows to be modified using a predicate. Those rows then have the update list applied to them. For example:

```
UPDATE foods SET name='CHOCOLATE BOBKA'
WHERE name='Chocolate Bobka';
SELECT * FROM foods WHERE name LIKE 'CHOCOLATE%';
```

```
id     type_  name
-----  -----  ------------------------------
10     1      CHOCOLATE BOBKA
11     1      Chocolate Eclairs
12     1      Chocolate Cream Pie
222    9      Chocolates, box of
223    9      Chocolate Chip Mint
224    9      Chocolate Covered Cherries
```

`UPDATE` is a very simple and direct command, and this is pretty much the extent of its use. As in `INSERT`, you must be aware of any `UNIQUE` constraints, as they will stop `UPDATE` every bit as much as `INSERT`:

```
sqlite> UPDATE foods SET id=11 where name='CHOCOLATE BOBKA';
SQL error: PRIMARY KEY must be unique
```

This is true for any constraint, however.

Deleting Records

You delete records from a table using the `DELETE` command. The `DELETE` command deletes rows from a table. `DELETE` has the general form

```
DELETE FROM table WHERE predicate;
```

Syntactically, `DELETE` is a watered-down `UPDATE` statement. Remove the `SET` clause from `UPDATE` and you have `DELETE`. The `WHERE` clause works exactly as in `SELECT`, except that it identifies rows to be deleted. For example:

```
DELETE FROM foods WHERE name='CHOCOLATE BOBKA';
```

Data Integrity

Data integrity is concerned with defining and protecting relationships within and between tables. There are four general types: *domain integrity*, *entity integrity*, *referential integrity*, and *user-defined integrity*. Domain integrity involves controlling values within columns. Entity integrity involves controlling rows in tables. Referential integrity involves controlling rows between tables—specifically foreign key relationships. And user-defined integrity is a catchall for everything else.

Data integrity is implemented using *constraints*. A constraint is a control measure used to restrict the values that can be stored in a column or columns. Going by just the values in columns, the database can enforce all four types of integrity constraints. In SQLite, constraints also include support for *conflict resolution*. Conflict resolution is covered in detail later in this chapter.

This section is a logical continuation of the "Creating a Database" section at the beginning of this chapter. As such, the examples in this section use the same `contacts` table defined there. It is listed again here for convenience:

```
CREATE TABLE contacts ( id INTEGER PRIMARY KEY,
                        name TEXT NOT NULL COLLATE NOCASE,
                        phone TEXT NOT NULL DEFAULT 'UNKNOWN',
                        UNIQUE (name,phone) );
```

As you know by now, constraints are a part of a table's definition. They can be associated with a column definition, or defined independently in the body of the table definition. Column-level constraints include `NOT NULL`, `UNIQUE`, `PRIMARY KEY`, `CHECK`, and `COLLATE`. Table-level constraints include `PRIMARY KEY`, `UNIQUE`, and `CHECK`. All of these constraints are covered in the following sections according to their respective integrity types.

The reason this material was not addressed earlier is because it requires familiarity with the `UPDATE`, `INSERT`, and `DELETE` commands. Just as these commands operate on data, constraints operate on them, making sure that they work within the guidelines defined in the tables they modify.

Entity Integrity

Entity integrity stems from the *Guaranteed Access Rule*, as explained in Chapter 3. It requires that every field in every table be addressable. That is, every field in the database must be uniquely identifiable and capable of being located. In order for a field to be addressable, its corresponding row must also be addressable. And for that, the row must be unique in some way. This is the job of the primary key.

The primary key consists of least one column, or a group of columns with a UNIQUE constraint. The UNIQUE constraint, as you will soon see, simply requires that every value in a column (or group of columns) be distinct. Therefore, the primary key ensures that each row is somehow distinct from all other rows in a table, ultimately ensuring that every field is also addressable. Entity integrity basically keeps data organized in a table. After all, what good is a field if you can't find it?

UNIQUE Constraints

Since primary keys are based on UNIQUE constraints, we'll start with them. A UNIQUE constraint simply requires that all values in a column or a group of columns are distinct from one another, or unique. If you attempt to insert a duplicate value, or update a value to another value that already exists in the column, the database will issue a constraint violation and abort the operation. UNIQUE constraints can be defined at the column or the table level. When defined at the table level, UNIQUE constraints can be applied across multiple columns. In this case, the combined value of the columns must be unique. In contacts, there is a unique constraint on both name and phone together. See what happens if I attempt to insert another 'Jerry' record with a phone value 'UNKNOWN':

```
sqlite> INSERT INTO contacts (name,phone) VALUES ('Jerry','UNKNOWN');
SQL error: columns name, phone are not unique

sqlite> INSERT INTO contacts (name) VALUES ('Jerry');
SQL error: columns name, phone are not unique

sqlite> INSERT INTO contacts (name,phone) VALUES ('Jerry', '555-1212');
```

In the first case, I explicitly specified name and phone. This matched the values of the existing record and the UNIQUE constraint kicked in and did not let me do it. The third INSERT illustrates that the UNIQUE constraint applies to name and phone combined, not individually. It inserted another row with 'Jerry' as the value for name, which did not cause an error, because name by itself it not unique—only name and phone together.

NULL AND UNIQUE

Pop quiz: Based on what you know about NULL and UNIQUE, how many NULL values can you put in a column that is declared UNIQUE? Answer: It depends on which database you are using. PostgreSQL and Oracle say as many as you want. Informix and Microsoft SQL Server say only one. DB2, SQL Anywhere, and Borland Inter-Base say none at all. SQLite follows Oracle and PostgreSQL—you can put as many NULLs as you want in a unique column. This is another classic case of NULL befuddling everyone. On one side, you can argue that one NULL value is never equal to another NULL value because you don't have enough information about either to know if they are equal. On the other side, you don't really have enough information to prove that they are different either. The consensus in SQLite is to assume that they are all different, so you can have a whole unique column stuffed full of NULLs if you like.

Primary Key Constraints

In SQLite, a primary key column is always defined when you create a table, whether you define one or not. This column is a 64-bit integer value called ROWID. It has two aliases, _ROWID_ and OID, which can be used to refer to it as well. Default values are automatically generated for it in monotonically increasing order.

SQLite provides an autoincrement feature for primary keys, should you want to define your own. If you define a column's type as INTEGER PRIMARY KEY, SQLite will create a DEFAULT value on that column, which will provide a monotonically increasing integer value that is guaranteed to be unique in that column. In reality, however, this column will simply be an alias for ROWID. They will all refer to the same value. Since SQLite uses a 64-bit number for the primary key, the maximum value for this column is 9,223,372,036,854,775,807.

■ **Note** You may be wondering where the maximum value of a key value comes from. It is based on the limits of a 64-bit integer. A 64-bit integer has 64 bits of storage that can be used to represent a number. That is, 2^{64} (or 18,446,744,073,709,551,616) possible values can be represented with 64 bits. Think of this as a range of values. An *unsigned* integer basically defines this range starting at 0. The range is therefore exclusively positive values, and therefore needs no sign to designate it—hence "unsigned." The maximum value of an unsigned 64-bit integer is therefore 18,446,744,073,709,551,615 (one less because the range starts at 0, not 1). However, SQLite uses *signed* integers. A signed integer splits the range so that half of it is less than zero and half is greater than zero. The range of a signed 64-bit integer is -9,223,372,036,854,775,808 to +9,223,372,036,854,775,807 (which is a range of 18,446,744,073,709,551,616 values). Key values in SQLite are signed 64-bit integers. Therefore, the maximum value of a key in SQLite is +9,223,372,036,854,775,807.

Even if you manage to reach this limit, SQLite will simply start searching for unique values that are not in the column for subsequent INSERTs. When you delete rows from the table, ROWIDs may be recycled and reused on subsequent INSERTs. As a result, newly created ROWIDs might not always be in strictly ascending order.

In the examples so far I have managed to insert two records into contacts. Not once did I ever specify a value for id. As mentioned before, this is because id is declared as INTEGER PRIMARY KEY. Therefore, SQLite supplied an incremental integer value for each INSERT automatically, as you can see here:

```
sqlite> SELECT * FROM contacts;

id   name   phone
---  -----  --------
1    Jerry  UNKNOWN
2    Jerry  555-1212
```

Notice that the primary key is accessible from all the aforementioned aliases, in addition to id:

```
sqlite> SELECT ROWID, OID,_ROWID_,id, name, phone FROM CONTACTS;
```

```
id  id  id  id  name   phone
--  --  --  --  ----   -----
1   1   1   1   Jerry  UNKNOWN
2   2   2   2   Jerry  555-1212
```

If you include the keyword AUTOINCREMENT after INTEGER PRIMARY KEY, SQLite will use a different key generation algorithm for the column. This algorithm basically prevents ROWIDs from being recycled. It guarantees that only new (not recycled) ROWIDs are provided for every INSERT. When a table is created with a column containing the AUTOINCREMENT constraint, SQLite will keep track of that column's maximum ROWID in a system table called sqlite_sequence. It will only use values greater than that maximum on all subsequent INSERTs. If you ever reach the absolute maximum, then SQLite will return a SQLITE_FULL error on subsequent INSERTs. For example:

```
sqlite> CREATE TABLE maxed_out(id INTEGER PRIMARY KEY AUTOINCREMENT, x text);
sqlite> INSERT INTO maxed_out VALUES (9223372036854775807, 'last one');
sqlite> SELECT * FROM sqlite_sequence;

name        seq
----------  -------------------
maxed_out   9223372036854775807

sqlite> INSERT INTO maxed_out VALUES (NULL, 'wont work');
SQL error: database is full
```

Here, I provided the primary key value. SQLite then stored this value as the maximum for maxed_out.id in sqlite_sequence. I supplied the very last (maximum) 64-bit value before wrap-around. In the next INSERT, I used the generated default value, which must be a monotonically increasing value. This wrapped around to 0, and SQLite issued a SQLITE_FULL error.

While SQLite tracks the maximum value for an AUTOINCREMENT column in the sqlite_sequence table, it does not prevent you from providing your own values for it in the INSERT command. The only requirement is that the value you provide must be unique within the column. For example:

```
sqlite> DROP TABLE maxed_out;
sqlite> CREATE TABLE maxed_out(id INTEGER PRIMARY KEY AUTOINCREMENT, x text);
sqlite> INSERT INTO maxed_out values(10, 'works');
sqlite> SELECT * FROM sqlite_sequence;

name        seq
----------  ----------
maxed_out   10

sqlite> INSERT INTO maxed_out values(9, 'works');
sqlite> SELECT * FROM sqlite_sequence;

name        seq
----------  ----------
maxed_out   10
```

```
sqlite> INSERT INTO maxed_out VALUES (9, 'fails');
SQL error: PRIMARY KEY must be unique

sqlite> INSERT INTO maxed_out VALUES (NULL, 'should be 11');
sqlite> SELECT * FROM maxed_out;

id          x
----------  ------------
9           works
10          works
11          should be 11

sqlite> SELECT * FROM sqlite_sequence;

name        seq
----------  ----------
maxed_out   11
```

Here, I dropped and re-created the maxed_out table, and inserted a record with an explicitly defined ROWID of 10. Then I inserted a record with a ROWID less than 10, which worked. I tried it again with the same value and it failed, due to the UNIQUE constraint. Finally, I inserted another record using the default key value, and SQLite provided the next monotonically increasing value—10+1.

In summary, AUTOINCREMENT prevents SQLite from recycling primary key values (ROWIDs) and stops when the ROWID reaches the maximum (signed) 64-bit integer value. This feature was added for specific applications that required this behavior. Unless you have such a specific need in your application, it is perhaps best to just use INTEGER PRIMARY KEY for autoincrement columns.

Like UNIQUE constraints, PRIMARY KEY constraints can be defined over multiple columns. You don't have to use an integer value for your primary key. If you choose to use another value, SQLite will still maintain the ROWID column internally, but it will also place a UNIQUE constraint on your declared primary key. For example:

```
sqlite> CREATE TABLE pkey(x text, y text, PRIMARY KEY(x,y));
sqlite> INSERT INTO pkey VALUES ('x','y');
sqlite> INSERT INTO pkey VALUES ('x','x');
sqlite> SELECT ROWID, x, y FROM pkey;

rowid       x           y
----------  ----------  ----------
1           x           y
2           x           x

sqlite> INSERT INTO pkey VALUES ('x','x');
SQL error: columns x, y are not unique
```

The primary key here is technically just a UNIQUE constraint across two columns, nothing more. As stated before, the concept of a primary key is more or less just lip service to the relational model—SQLite always provides one whether you do or not. If you do define you own primary key, it is in reality just another UNIQUE constraint, nothing more.

Domain Integrity

The simplest definition of domain integrity is the conformance of a column's values to its assigned domain. That is, every value in a column should exist within that column's defined domain. However, the term *domain* is a little vague. Domains are often compared to types in programming languages, such as strings or floats. And while that is not a bad analogy, domain integrity is actually much broader than that.

Domain constraints make it possible for you to start with a simple type—like an integer—and add additional constraints to create a more restricted set of acceptable values for a column. For example, you can create a column with an integer type and add the constraint that only three such values are allowed: {-1, 0. 1}. In this case, you have modified the range of acceptable values (from the domain of all integers to just three integers), but not the data type itself. You are dealing with two things: a type and a range.

Consider another example: the name column in the contacts table. It is declared as follows:

```
name TEXT NOT NULL COLLATE NOCASE
```

The domain TEXT defines the type and initial range of acceptable values. Everything following it serves to restrict and qualify that range even further. The name column is then the domain of all TEXT values that do not include NULL values where uppercase letters and lowercase letters have equal value. It is still TEXT, and operates as TEXT, but its range of acceptable values is further restricted from that of TEXT.

You might say that a column's domain is not the same thing as its type. Rather, its domain is a combination of two things: a type and a range. The column's type defines the representation and operators of its values—how they are stored and how you can operate on them—sort, search, add, subtract, and so forth. A column's range is its set of acceptable values you can store in it, which is not necessarily the same as its declared type. The type's range represents a maximum range of values. The column's range—as you have seen—can be restricted through constraints. So for all practical purposes, you can think of a column's domain as a type with constraints tacked on.

Similarly, there are essentially two components to domain integrity: type checking and range checking. While SQLite supports many of the standard domain constraints for range checking (NOT NULL, CHECK, etc.), its approach to type checking is where things diverge from other databases. In fact, SQLite's approach to types and type checking is one of its most controversial, misunderstood, and disputed features.

But before we get into how SQLite handles types, let's cover the easy stuff first: default values, NOT NULL constraints, CHECK constraints, and collations.

Default Values

The DEFAULT keyword provides a default value for a column if one is not provided in an INSERT command. DEFAULT is not a constraint, because it doesn't enforce anything. It simply steps in when needed. However, it does fall within domain integrity because it provides a policy for handling NULL values in a column. If a column doesn't have a default value, and you don't provide a value for it in an INSERT statement, then SQLite will insert NULL for that column. For example, contacts.name has a default value of 'UNKNOWN'. With this in mind, consider the following example:

```
sqlite> INSERT INTO contacts (name) VALUES ('Jerry');
sqlite> SELECT * FROM contacts;

id          name        phone
----------  ----------  ----------
1           Jerry       UNKNOWN
```

The INSERT command inserted a row, specifying a value for name but not phone. As you can see from the resulting row, the default value for phone kicked in and provided the string 'UNKNOWN'. If phone did not have a default value, then in this example, the value for phone in this row would have been NULL instead.

DEFAULT also accepts three predefined ANSI/ISO reserved words for generating default dates and times. CURRENT_TIME will generate the current local time in ANSI/ISO time format (HH:MM:SS). CURRENT_DATE will generate the current date (in YYYY-MM-DD format). CURRENT_TIMESTAMP will produce a combination of these two (in YYYY-MM-DD HH:MM:SS format). For example:

```
CREATE TABLE times ( id int,
  time NOT NULL DEFAULT CURRENT_DATE
  time NOT NULL DEFAULT CURRENT_TIME,
  time NOT NULL DEFAULT CURRENT_TIMESTAMP );
INSERT INTO times(1);
INSERT INTO times(2);
SELECT * FROMS times;
```

```
id   date        time        timestamp
---  ----------  ----------  -------------------
1    2006-03-15  23:30:25    2006-03-15 23:30:25
2    2006-03-15  23:30:40    2006-03-15 23:30:40
```

These defaults come in quite handy for tables that need to log or timestamp events.

NOT NULL Constraints

If you are one of those people who are not fond of NULL, then the NOT NULL constraint is for you. NOT NULL ensures that values in the column may never be NULL. INSERT commands may not add NULL in the column, and UPDATE commands may not change existing values to NULL. Oftentimes, you will see NOT NULL raise its ugly head in INSERT statements. Specifically, a NOT NULL constraint without a DEFAULT constraint will prevent any unspecified values from being used in the INSERT (because the default values provided in this case are NULL). In the preceding example, the NOT NULL constraint on name requires that an INSERT command always provide a value for that column. For example:

```
sqlite> INSERT INTO contacts (phone) VALUES ('555-1212');
SQL error: contacts.name may not be NULL
```

This INSERT command specified a phone value, but not a name. The NOT NULL constraint on name kicked in and forbade the operation.

The way to shut NOT NULL up is to also include a DEFAULT constraint in a column. This is the case for phone. While phone has a NOT NULL constraint, it has a DEFAULT constraint as well. If an INSERT command does not specify a value for phone, the DEFAULT constraint steps in and provides the value 'UNKNOWN', thus satisfying the NOT NULL constraint. To this end, people often use DEFAULT constraints in conjunction with NOT NULL constraints so that INSERT commands can safely use default values while at the same time keeping NULL out of the column.

CHECK Constraints

CHECK constraints allow you to define expressions to test values whenever they are inserted into or updated within a column. If the values do not meet the criteria set forth in the expression, the database issues a constraint violation. Thus, it allows you to define additional data integrity checks beyond UNIQUE or NOT NULL to suit your specific application. An example of a CHECK constraint might be to ensure that the value of a phone number field is at least seven characters long. To do this, you can either add the constraint to the column definition of phone, or as a standalone constraint in the table definition as follows:

```
CREATE TABLE contacts ( id INTEGER PRIMARY KEY,
                        name TEXT NOT NULL COLLATE NOCASE,
                        phone TEXT NOT NULL DEFAULT 'UNKNOWN',
                        UNIQUE (name,phone),
                        CHECK(LENGTH(phone)>=7) );
```

Here, any attempt to insert or update a value for phone less than seven characters will result in a constraint violation. You can use any expression in a CHECK constraint that you would in a WHERE clause. For example, say you have the table foo defined as follows:

```
CREATE TABLE foo( x integer,
                  y integer CHECK(y>x),
                  z integer CHECK(z>ABS(y)) );
```

In this table, every value of z must always be greater than y, which in turn must be greater than x. To show illustrate this, try the following:

```
INSERT into foo values (-2, -1, 2);
INSERT into foo values (-2, -1, 1);
SQL error: constraint failed

UPDATE foo SET y=-3 WHERE x=-3;
SQL error: constraint failed
```

The CHECK constraints for all columns are evaluated before any modification is made. For the modification to succeed, the expressions for all constraints must evaluate to true.

Functionally, triggers can be used just as effectively as check constraints for data integrity. In fact, triggers can do much more. If you find that you can't quite express what you need in a CHECK constraint, then triggers are a good alternative. Triggers are covered later in this chapter in the section "Triggers."

Collations

Collation is related to domain integrity in that it defines what constitutes unique text values. Collation specifically refers to how text values are compared. Different collations employ different comparison methods. For example, one collation might be case insensitive, so the strings `'JujyFruit'` and `'JUJYFRUIT'` are considered the same. Another collation might be case sensitive, in which case the strings would be considered different.

SQLite has three built-in collations. The default is `BINARY`, which compares text values byte by byte using a specific C function called `memcmp()`. This happens to work nicely for many Western languages such as English. `NOCASE` is basically a case-insensitive collation for the 26 ASCII characters used in English. Finally there is `REVERSE`, which is the reverse of the `BINARY` collation. `REVERSE` is more for testing (and perhaps illustration) than anything else.

The SQLite C API provides a way to create custom collations. This feature allows developers to support languages and/or locales that are not well served by the `BINARY` collation. See Chapter 7 for more information.

The `COLLATE` keyword defines the collation for a column. For example, the collation for `contacts.name` is defined as `NOCASE`, which means that it is case insensitive. Thus, if I try to insert another row with a `name` value of `'JERRY'` and a `phone` value of `'555-1212'` it should fail:

```
sqlite> INSERT INTO contacts (name,phone) VALUES ('JERRY','555-1212');
SQL error: columns name, phone are not unique
```

According to `name`'s collation, `'JERRY'` is the same as `'Jerry'`, and there is already a row with that value. Therefore, a new row with `name='JERRY'` would be a duplicate value. By default, collation in SQLite is case sensitive. The previous example would have worked had I not defined `NOCASE` on `name`.

Storage Classes

As mentioned earlier, SQLite does not work like other databases when it comes to handling data types. It differs in the types it supports, and in how they are stored, compared, enforced, and assigned. The following sections explore SQLite's radically different but surprisingly flexible approach to data types and its relation to domain integrity.

With respect to types, SQLite's domain integrity is better described as *domain affinity*. In SQLite, it is referred to as *type affinity*. To understand type affinity, however, you must first understand *storage classes* and something called *manifest typing*.

Internally, SQLite has five primitive data types, which are referred to as *storage classes*. The term storage class refers to the format in which a value is stored on disk. Regardless, it is still synonymous with type, or data type. The five storage classes are described in Table 4-6.

Table 4-6. *SQLite Storage Classes*

Name	Description
`INTEGER`	Integer values are whole numbers (positive and negative). They can vary in size: 1, 2, 3, 4, 6, or 8 bytes. The maximum integer range (8 bytes) is {-9223372036854775808,-1,0,1, -9223372036854775807}. SQLite automatically handles the integer sizes based on the numeric value.
`REAL`	Real values are real numbers with decimal values. SQLite uses 8-byte floats to store real numbers.
`TEXT`	Text is character data. SQLite supports various character encodings, which include UTF-8 and UTF-16 (big and little endian). The maximum string value in SQLite is unlimited.
`BLOB`	Binary large object (`BLOB`) data is any kind of data. The maximum size for `BLOB`s in SQLite is unlimited.
`NULL`	`NULL` values represent missing information. SQLite has full support for `NULL` values.

SQLite infers a value's type from its representation. The following inference rules are used to do this:

- A value specified as a literal in SQL statements is assigned class `TEXT` if it is enclosed by single or double quotes.
- A value is assigned class `INTEGER` if the literal is specified as an unquoted number with no decimal point or exponent.
- A value is assigned class `REAL` if the literal is an unquoted number with a decimal point or an exponent.
- A value is assigned class `NULL` if its value is `NULL`.
- A value is assigned class `BLOB` if it is of the format `X'ABCD'`, where `ABCD` are hexadecimal numbers. The `X` prefix and values can be either uppercase or lowercase.

The `typeof()` SQL function returns the storage class of a value based on its representation. Using this function, the following SQL illustrates type inference in action:

```
sqlite> select typeof(3.14), typeof('3.14'),
        typeof(314), typeof(x'3142'), typeof(NULL);

typeof(3.14)  typeof('3.14')  typeof(314)  typeof(x'3142')  typeof(NULL)
------------  --------------  -----------  ---------------  ------------
real          text            integer      blob             null
```

Here are all of the five internal storage classes invoked by specific representations of data. The value 3.14 looks like a `REAL`, and therefore is a `REAL`. The value `'3.14'` looks like `TEXT`, and therefore is `TEXT`, and so on.

A single column in SQLite may contain different values *of different storage classes*. Consider the following example:

```
sqlite> DROP TABLE domain;
sqlite> CREATE TABLE domain(x);
sqlite> INSERT INTO domain VALUES (3.142);
sqlite> INSERT INTO domain VALUES ('3.142');
sqlite> INSERT INTO domain VALUES (3142);
sqlite> INSERT INTO domain VALUES (x'3142');
sqlite> INSERT INTO domain VALUES (NULL);
sqlite> SELECT ROWID, x, typeof(x) FROM domain;

rowid       x           typeof(x)
----------  ----------  ----------
1           3.142       real
2           3.142       text
3           3142        integer
4           1B          blob
5           NULL        null
```

This raises a few questions. How are the values in a column sorted or compared? How do you sort a column with `INTEGER`, `REAL`, `TEXT`, `BLOB`, and `NULL` values? How do you compare an `INTEGER` with a `BLOB`? Which is greater? Can they ever be equal?

As it turns out, values in a column with different storages classes can be sorted. And they can be sorted because they can be compared. There are well-defined rules to do so. Storage classes are sorted by using their respective *class values*, which are defined as follows:

1. The `NULL` storage class has the lowest class value. A value with a `NULL` storage class is considered less than any other value (including another value with storage class `NULL`). Within `NULL` values, there is no specific sort order.

2. `INTEGER` or `REAL` storage classes have higher value than `NULL`s, and share equal class value. `INTEGER` and `REAL` values are compared numerically.

3. The `TEXT` storage class has higher value than `INTEGER` or `REAL`. A value with an `INTEGER` or a `REAL` storage class will always be less than a value with a `TEXT` storage class no matter its value. When two `TEXT` values are compared, the comparison is determined by the collation defined for the values.

4. The `BLOB` storage class has the highest value. Any value that is not of class `BLOB` will always be less than a value of class `BLOB`. `BLOB` values are compared using the C function `memcmp()`.

So when SQLite sorts a column, it first groups values according to storage class—first `NULL`s, then `INTEGER`s and `REAL`s, next `TEXT`, and finally `BLOB`s. It then sorts the values within each group. `NULL`s are not ordered at all, `INTEGER`s and `REAL`s are compared numerically, `TEXT` is arranged by the appropriate collation, and `BLOB`s are sorted using `memcmp()`. Figure 4-20 illustrates a hypothetical column sorted in ascending order.

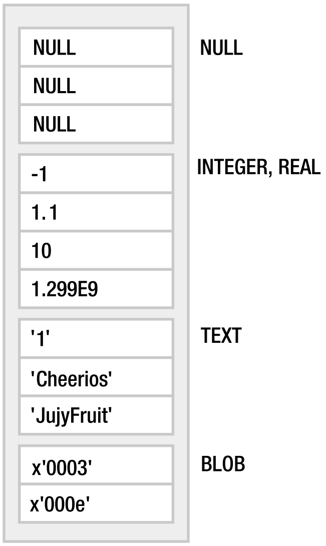


Figure 4-20. *Storage class sort order*

The following SQL illustrates the differences between storage class values:

```
sqlite> SELECT 3 < 3.142, 3.142 < '3.142', '3.142' < x'3000',
               x'3000' < x'3001';

3 < 3.142  3.142 < '3.142' '3.142' < x'3000' x'3000' < x'3001'
---------  --------------- ----------------- ------------------
1          1               1                 1
```

`INTEGER`s and `REAL`s are compared numerically and are both less than `TEXT`s, and `TEXT`s are less than `BLOB`s.

Manifest Typing

SQLite uses manifest typing. If you do a little research, you will find that the term *manifest typing* is subject to multiple interpretations. In programming languages, manifest typing refers to how the type of a variable or value is defined and/or determined. There are two main interpretations:

> *Manifest typing means that a variable's type must be explicitly declared in the code.* By this definition, languages such as C/C++, Pascal, and Java would be said to use manifest typing. Dynamically typed languages such as Perl, Python, and Ruby, on the other hand, are the direct opposite as they do not require that a variable's type be declared.
>
> *Manifest typing means that variables don't have types at all. Rather, only values have types.* This seems to be in line with dynamically typed languages. Basically, a variable can hold any value, and the type of that variable at any point in time is determined by its value at that moment. Thus if you set variable `x=1`, then `x` at that moment is of type `INTEGER`. If you then set `x='JujyFruit'`, it is then of type `TEXT`. That is, if it looks like an `INTEGER`, and it acts like an `INTEGER`, it is an `INTEGER`.

For the sake of brevity, I will refer to the first interpretation as MT 1 and the second as MT 2. At first glance, it may not be readily apparent as to which interpretation best fits SQLite. For example, consider the following table:

```
CREATE TABLE foo( x integer,
                  y text,
                  z real );
```

Say we now insert a record into this table as follows:

```
INSERT INTO foo VALUES ('1', '1', '1');
```

When SQLite creates the record, what type is stored internally for x, y, and z? The answer: INTEGER, TEXT, and REAL. Then it seems that SQLite uses MT 1: variables have declared types. But wait a second; column types in SQLite are optional, so we could have just as easily defined foo as follows:

```
CREATE TABLE foo(x, y, z);
```

Now let's do the same INSERT:

```
INSERT INTO foo VALUES ('1', '1', '1');
```

What type are x, y, and z now? The answer: TEXT, TEXT, and TEXT. Well, maybe SQLite is just setting columns to TEXT by default. If you think that, then consider the following INSERT statement on the same table:

```
INSERT INTO foo VALUES (1, 1.0, x'10');
```

What are x, y, and z in this row? INTEGER, REAL, and BLOB. This looks like MT 2, where the value itself determines its type.

So which one is it? The short answer: neither and both. The long answer is a little more involved. With respect to MT 1, SQLite lets you declare columns to have types if you want to. This looks and feels like what other databases do. But you don't *have to*, thereby violating this interpretation as well. This is because in all situations SQLite can take any value and infer a type from it. It doesn't need the type declared in the column to help it out. With respect to MT 2, SQLite allows the type of the value to "influence" (maybe not completely determine) the type that gets stored in the column. But you can still declare the column with a type and that type will exert some influence, thereby violating this interpretation as well—that types come from values only. What we really have here is the MT 3—the SQLite interpretation. It borrows from both MT 1 and MT 2.

But interestingly enough, manifest typing does not address the whole issue with respect to types. It seems to be concerned with only declaring and resolving types. What about type checking? That is, if you declare a column to be type integer, what exactly does that mean?

First let's consider what most other relational databases do. They enforce *strict type checking* as a standard part of standard domain integrity. First you declare a column's type. Then only values of that type can go in it. End of story. You can use additional domain constraints if you want, but under no conditions can you ever insert values of other types. Consider the following example with Oracle:

```
SQL> create table domain(x int, y varchar(2));
Table created.

SQL> INSERT INTO domain VALUES ('pi', 3.14);
INSERT INTO domain VALUES ('pi', 3.14)
                         *
ERROR at line 1:
ORA-01722: invalid number
```

The value `'pi'` is not an integer value. And column `x` was declared to be of type `int`. I don't even get to hear about the error in column `y` because the whole `INSERT` is aborted due to the integrity violation on `x`. When I try this in SQLite,

```
sqlite> CREATE TABLE domain (x int, y varchar(2));
sqlite> INSERT INTO domain VALUES ('pi', 3.14);
sqlite> SELECT * FROM domain;

x     y
----  -----
pi    3.14
```

there's no problem. I said one thing and did another, and SQLite didn't stop me. *SQLite's domain integrity does not include strict type checking.* So what is going on? Does a column's declared type count for anything? Yes. Then how is it used? It is all done with something called *type affinity*.

In short, SQLite's manifest typing states that 1) columns can have types and 2) that types can be inferred from values. Type affinity addresses how these two things relate to one another. Type affinity is a delicate balancing act that sits between strict typing and dynamic typing.

Type Affinity

In SQLite, columns don't have types or domains. While a column can have a declared type, internally it only has a type affinity. Declared type and type affinity are two different things. Type affinity determines the storage class SQLite uses to store values within a column. The actual storage class a column uses to store a given value is a function of both the value's storage class and the column's affinity. Before getting into how this is done, however, let's first talk about how a column gets its affinity.

Column Types and Affinities

To begin with, every column has an affinity. There are four different kinds: `NUMERIC`, `INTEGER`, `TEXT`, and `NONE`. A column's affinity is determined directly from its declared type (or lack thereof). Therefore, when you declare a column in a table, the type you choose to declare it as will ultimately determine that column's affinity. SQLite assigns a column's affinity according to the following rules:

- By default, a column's default affinity is `NUMERIC`. That is, if a column is not `INTEGER`, `TEXT`, or `NONE`, then it is automatically assigned `NUMERIC` affinity.
- If a column's declared type contains the string `'INT'` (in uppercase or lowercase), then the column is assigned `INTEGER` affinity.

- If a column's declared type contains any of the strings `'CHAR'`, `'CLOB'`, or `'TEXT'` (in uppercase or lowercase), then that column is assigned `TEXT` affinity. Notice that `'VARCHAR'` contains the string `'CHAR'` and thus will confer `TEXT` affinity.
- If a column's declared type contains the string `'BLOB'` (in uppercase or lowercase), *or if it has no declared type*, then it is assigned `NONE` affinity.

■ **Note** Pay attention to defaults. If you don't declare a column's type, then its affinity will be `NONE`, in which case all values will be stored using their given storage class (or inferred from their representation). If you are not sure what you want to put in a column, or want to leave it open to change, this is the best affinity to use. However, be careful of the scenario where you declare a type that does not match any of the rules for `NONE`, `TEXT`, or `INTEGER`. While you might intuitively think the default should be `NONE`, it is actually `NUMERIC`. For example, if you declare a column of type `JUJYFRUIT`, it will *not* have affinity `NONE` just because SQLite doesn't recognize it. Rather it will have affinity `NUMERIC`. (Interestingly, the scenario also happens when you declare a column's type to be `numeric` for the same reason.) Rather than using an unrecognized type that ends up as `numeric`, you may prefer to leave the column's declared type out altogether, which will ensure it has affinity `NONE`.

Affinities and Storage

Each affinity influences how values are stored in its associated column. The rules governing storage are as follows:

- A `NUMERIC` column may contain all five storage classes. A `NUMERIC` column has a bias toward numeric storage classes (`INTEGER` and `REAL`). When a `TEXT` value is inserted into a `NUMERIC` column, it will attempt to convert it to an `INTEGER` storage class. If this fails, it will attempt to convert it to a `REAL` storage class. Failing that, it stores the value using the `TEXT` storage class.
- An `INTEGER` column tries to be as much like a `NUMERIC` column as it can. An `INTEGER` column will store a `REAL` value as `REAL`. *However*, if a `REAL` value has *no fractional component*, then it will be stored using an `INTEGER` storage class. `INTEGER` column will try to store a `TEXT` value as `REAL` if possible. If that fails, they try to store it as `INTEGER`. Failing that, `TEXT` values are stored as `TEXT`.
- A `TEXT` column will convert all `INTEGER` or `REAL` values to `TEXT`.
- A `NONE` column does not attempt to convert any values. All values are stored using their given storage class.
- No column will ever try to convert `NULL` or `BLOB` values—regardless of affinity. `NULL` and `BLOB` values are always stored as is in every column.

These rules may initially appear somewhat complex, but their overall design goal is simple: to make it possible for SQLite to mimic other relational databases if you need it to. That is, if you treat columns like a traditional database, type affinity rules will store values in the way you expect. If you declare a column of type `INTEGER`, and put integers into it, they will be stored as

INTEGER. If you declare a column to be of type TEXT, CHAR, or VARCHAR and put integers into it, they will be stored as TEXT. However, if you don't follow these conventions, SQLite will still find a way to store the value.

Affinities in Action

Let's look at a few examples to get the hang of how affinity works. Consider the following:

```
sqlite> CREATE TABLE domain(i int, n numeric, t text, b blob);
sqlite> INSERT INTO domain VALUES (3.142,3.142,3.142,3.142);
sqlite> INSERT INTO domain VALUES ('3.142','3.142','3.142','3.142');
sqlite> INSERT INTO domain VALUES (3142,3142,3142,3142);
sqlite> INSERT INTO domain VALUES (x'3142',x'3142',x'3142',x'3142');
sqlite> INSERT INTO domain VALUES (null,null,null,null);
sqlite> SELECT ROWID,typeof(i),typeof(n),typeof(t),typeof(b) FROM domain;

rowid       typeof(i)   typeof(n)   typeof(t)   typeof(b)
----------  ----------  ----------  ----------  ----------
1           real        real        text        real
2           real        real        text        text
3           integer     integer     text        integer
4           blob        blob        blob        blob
5           null        null        null        null
```

The first INSERT inserts a REAL value. You can see this both by the format in the INSERT statement and by the resulting type shown in the typeof(b) column returned in the SELECT statement. Remember that BLOB columns have storage class NONE, which does not attempt to convert the storage class of the input value, so column b uses the same storage class that was defined in the INSERT statement. Column i keeps the NUMERIC storage class, because it tries to be NUMERIC when it can. Column n doesn't have to convert anything. Column t converts it to TEXT. Column b stores it exactly as given in the context. In each subsequent INSERT, you can see how the conversion rules are applied in each varying case.

The following SQL illustrates storage class sort order and interclass comparison (which are governed by the same set of rules):

```
sqlite> SELECT ROWID, b, typeof(b) FROM domain ORDER BY b;

rowid  b       typeof(b)
-----  ------  ---------
5      NULL    null
1      3.142   real
3      3142    integer
2      3.142   text
4      1B      blob
```

Here, you see that NULLs sort first, followed by INTEGERs and REALs, followed by TEXTs, then BLOBs. The following SQL shows how these values compare with the integer 1,000. The INTEGER and REAL values in b are less than 1,000 because they are numerically compared, while TEXT and BLOB are greater than 1,000 because they are in a higher storage class.

```
sqlite> SELECT ROWID, b, typeof(b), b<1000 FROM domain ORDER BY b;

rowid  b      typeof(b)  b<1000
-----  -----  --------  ----------
5      NULL   null      NULL
1      3.142  real      1
3      3142   integer   1
2      3.142  text      0
4      1B     blob      0
```

The primary difference between type affinity and strict typing is that type affinity will never issue a constraint violation for incompatible data types. SQLite will always find a data type to put any value into any column. The only question is what type it will use to do so. The only role of a column's declared type in SQLite is simply to determine its affinity. Ultimately, it is the column's affinity that has any bearing on how values are stored inside of it. However, SQLite does provide facilities for ensuring that a column may only accept a given type, or range of types. You do this using `CHECK` constraints, explained in the sidebar "Makeshift Strict Typing," later in this section.

Storage Classes and Type Conversions

Another thing to note about storage classes is that they can sometimes influence how values are compared as well. Specifically, SQLite will sometimes convert values between numeric storage classes (`INTEGER` and `REAL`) and `TEXT` before comparing them. For binary comparisons, it uses the following rules:

- When a column value is compared to the result of an expression, the affinity of the column is applied to the result of the expression before the comparison takes place.
- When two column values are compared, if one column has `INTEGER` or `NUMERIC` affinity and the other doesn't, then `NUMERIC` affinity is applied to `TEXT` values in the non-`NUMERIC` column.
- When two expressions are compared, SQLite does not make any conversions. The results are compared as is. If the expressions are of like storage class, then the comparison function associated with that storage class is used to compare values. Otherwise, they are compared on the basis of their storage class.

Note that the term *expression* here refers to any scalar expression or literal *other than a column value*. To illustrate the first rule, consider the following:

```
sqlite> select ROWID,b,typeof(i),i>'2.9' from domain ORDER BY b;

rowid  b      typeof(i  i>'2.9'
-----  -----  --------  ------------
5      NULL   null      NULL
1      3.142  real      1
3      3142   integer   1
2      3.142  real      1
4      1B     blob      1
```

The expression '2.9', while being TEXT, is converted to INTEGER before the comparison. So the column interprets the value in light of what it is. What if '2.9'was a non-numeric string? Then SQLite falls back to comparing storage class, in which INTEGER and NUMERIC types are always less than TEXT:

```
sqlite> SELECT ROWID,b,typeof(i),i>'text' FROM domain ORDER BY b;

rowid  b      typeof(i  i>'text'
-----  -----  --------  ------------
5      NULL   null      NULL
1      3.14   real      0
3      314    integer   0
2      3.14   real      0
4      1B     blob      1
```

The second rule simply states that when comparing a numeric and non-numeric column, where possible SQLite will try to convert the non-numeric column to numeric format:

```
sqlite> CREATE TABLE rule2(a int, b text);
sqlite> insert into rule2 values(2,'1');
sqlite> insert into rule2 values(2,'text');
sqlite> select a, typeof(a),b,typeof(b), a>b from rule2;

a           typeof(a)   b           typeof(b)   a>b
----------  ----------  ----------  ----------  ----------
2           integer     1           text        1
2           integer     text        text        0
```

Column a is an INTEGER, b is TEXT. When evaluating the expression a>b, SQLite tries to coerce b to INTEGER where it can. In the first row, b is '1', which can be coerced to INTEGER. SQLite makes the conversion and compares integers. In the second row, b is 'text' and can't be converted. SQLite then compares storage classes INTEGER and TEXT.

The third rule just reiterates that storage classes established by context are compared at face value. If what looks like a TEXT type is compared with what looks like an INTEGER type, then TEXT is greater.

Additionally, you can manually convert the storage type of a column or an expression using the CAST function. Consider the following example:

```
sqlite> SELECT typeof(3.14), typeof(CAST(3.14 as TEXT));

typeof(3.14)  typeof(CAST(3.14 as TEXT))
------------  --------------------------
real          text
```

MAKESHIFT STRICT TYPING

If you need something stronger than type affinity for domain integrity, then CHECK constraints can help. You can implement pseudo strict typing directly using a single built-in function and a CHECK constraint. As mentioned earlier, SQLite has a function which returns the inferred storage class of a value—typeof(). You can use typeof() in any relational expression to test for a values type. For example:

```
sqlite> select typeof(3.14) = 'text';
0
sqlite> select typeof(3.14) = 'integer';
0
sqlite> select typeof(3.14) = 'real';
1
sqlite> select typeof(3) = 'integer';
1
sqlite> select typeof('3') = 'text';
1
```

Therefore, you can use this function to implement a CHECK constraint that limits the acceptable types allowed in a column:

```
sqlite> create table domain (x integer CHECK(typeof(x)='integer'));
sqlite> INSERT INTO domain VALUES('1');
SQL error: constraint failed

sqlite> INSERT INTO domain VALUES(1.1);
SQL error: constraint failed

sqlite> INSERT INTO domain VALUES(1);
sqlite> select x, typeof(x) from domain;

x   typeof(x)
--  ----------
1   integer
sqlite> update domain set x=1.1;
SQL error: constraint failed
```

The only catch here is that you are limited to checking for SQLite’s native storage classes (or what can be implemented using other built-in SQL functions). However, if you are a programmer and either use a language extension that supports SQLite’s user-defined functions (e.g., Perl, Python, or Ruby) or use the SQLite C API directly, you can implement even more elaborate functions for type checking, which can be called from within CHECK constraints. Chapter 5 covers this in more detail.

Transactions

Transactions define boundaries around a group of SQL commands such that they either all successfully execute together or not at all. A classic example of the rationale behind transactions is a money transfer. Say a bank program is transferring money from one account to another. The money transfer program can do this in one of two ways: first insert (credit) the funds into account 2 then delete (debit) it from account 1, or first delete it from account 1 and insert it into account 2. Either way, the transfer is a two-step process: an `INSERT` followed by a `DELETE`, or a `DELETE` followed by an `INSERT`.

Now, say the program is in the process of making a transfer. The first SQL command completes successfully, and then the database server suddenly crashes or the power goes out. Whatever the case, the second operation does not complete. Now the money either exists in both accounts (the first scenario) or has been completely lost altogether (second scenario). Either way, someone's not going to be happy. And the database is in an inconsistent state.

The point here is that these two operations must either happen together or not at all. That is what transactions are for. Now let's replay the example with transactions. In the new scenario, the program first starts a transaction in the database, completes the first SQL operation, and then the lights go out. When they come back on and the database comes back up, it sees an incomplete transaction. It then undoes the changes of the first SQL operation, which brings it back into a consistent state—back where it started before the transfer.

Transaction Scopes

Transactions are issued with three commands: `BEGIN`, `COMMIT`, and `ROLLBACK`. `BEGIN` starts a transaction. Every operation following a `BEGIN` can be potentially undone, and will be undone if a `COMMIT` is not issued before the session terminates. The `COMMIT` command commits the work performed by all operations since the start of the transaction. Similarly, the `ROLLBACK` command undoes all of the work performed by all operations since the start of the transaction. A transaction is a scope in which operations are performed together and committed, or completely reversed. Consider the following example:

```
sqlite> BEGIN;
sqlite> DELETE FROM foods;
sqlite> ROLLBACK;
sqlite> SELECT COUNT(*) FROM foods;

COUNT(*)
--------
412
```

I started a transaction, deleted all the rows in `foods`, changed my mind, and reversed those changes by issuing a `ROLLBACK`. The `SELECT` statement shows that nothing was changed.

By default, every SQL command in SQLite is run under its own transaction. That is, if you do not define a transaction scope with `BEGIN...COMMIT/ROLLBACK`, SQLite will implicitly wrap every individual SQL command with a `BEGIN...COMMIT/ROLLBACK`. In that case, every command that completes successfully is committed. Likewise, every command that encounters an error is rolled back. This mode of operation (implicit transactions) is referred to as *autocommit mode*:

SQLite automatically runs each command in its own transaction, and if the command does not fail, its changes are automatically committed.

Conflict Resolution

As you've seen in previous examples, constraint violations cause the command that committed the violation to terminate. What exactly happens when a command terminates in the middle of making a bunch of changes to the database? In most databases, all of the changes are undone. That is the way the database is programmed to handle a constraint violation—end of story.

SQLite, however, has a unique feature that allows you to specify different ways to handle (or recover from) constraint violations. It is called *conflict resolution*. Take, for example, the following `UPDATE`:

```
sqlite> UPDATE foods SET id=800-id;
SQL error: PRIMARY KEY must be unique
```

This results in a `UNIQUE` constraint violation because once the `UPDATE` statement reaches the 388th record, it attempts to update its `id` value to 800-388=412. But a row with an `id` of 412 already exists, so it aborts the command. But SQLite already updated the first 387 rows before it reached this constraint violation. What happens to them? The default behavior is to terminate the command and reverse all of the changes it made, while leaving the transaction intact.

But what if you wanted these 387 changes to stick despite the constraint violation? Well, believe it or not, you can have it that way too, if you want. You just need to use the appropriate conflict resolution. There are five possible resolutions, or policies, that can be applied to address a conflict (constraint violation): `REPLACE`, `IGNORE`, `FAIL`, `ABORT`, and `ROLLBACK`. These five resolutions define a spectrum of error tolerance or sensitivity. On one end of the spectrum is `REPLACE`, which will effectively allow a statement to plow through almost every possible constraint violation. On the other end is `ROLLBACK`, which will terminate the entire transaction upon the first violation of any kind. The resolutions are defined as follows in order of their severity:

- `REPLACE`: When a `UNIQUE` constraint violation is encountered, SQLite removes the row (or rows) that caused the violation, and replaces it (them) with the new row from the `INSERT` or `UPDATE`. The SQL operation continues without error. If a `NOT NULL` constraint violation occurs, the `NULL` value is replaced by the default value for that column. If the column has no default value, then SQLite applies the `ABORT` policy. It is important to note that when this conflict resolution strategy deletes rows in order to satisfy a constraint, it does not invoke delete triggers on those rows. This behavior, however, is subject to change in a future release.

- `IGNORE`: When a constraint violation is encountered, SQLite allows the command to continue, and leaves the row that triggered the violation unchanged. Other rows before and after the row in question continue to be modified by the command. Thus, all rows in the operation that trigger constraint violations are simply left unchanged, and the command proceeds without error.

- FAIL: When a constraint violation is encountered, SQLite terminates the command but does not restore the changes it made prior to encountering the violation. That is, all changes within the SQL command up to the violation are preserved. For example, if an UPDATE statement encountered a constraint violation on the 100th row it attempts to update, then the changes to the first 99 rows already modified remain intact, but changes to rows 100 and beyond never occur as the command is terminated.
- ABORT: When a constraint violation is encountered, SQLite restores all changes the command made and terminates it. ABORT is the default resolution for all operations in SQLite. It is also the behavior defined in the SQL standard. As a side note, ABORT is also the most expensive conflict resolution policy—requiring extra work even if no conflicts ever occur.
- ROLLBACK: When a constraint violation is encountered, SQLite performs a ROLLBACK—aborting the current command along with the entire transaction. The net result is that all changes made by the current command *and all previous commands* in the transaction are rolled back. This is the most drastic level of conflict resolution where a single violation results in a complete reversal of everything performed in a transaction.

Conflict resolution can be specified within SQL commands as well as within table and index definitions. Specifically, conflict resolution can be specified in INSERT, UPDATE, CREATE TABLE, and CREATE INDEX. Furthermore, it has specific implications within triggers. The syntax for conflict resolution in INSERT and UPDATE is as follows:

```
INSERT OR resolution INTO table (column_list) VALUES (value_list);
UPDATE OR resolution table SET (value_list) WHERE predicate;
```

The conflict resolution policy comes right after the INSERT or UPDATE command and is prefixed with OR. Also, the INSERT OR REPLACE expression can be abbreviated as just REPLACE.

In the preceding UPDATE example, the updates made to the 387 records were rolled back because the default resolution is ABORT. If you wanted the updates to stick, you could use the FAIL resolution. To illustrate this, in the following example I copy foods into a new table test and use it as the guinea pig. I add an additional column to test called modified, the default value of which is 'no'. In the UPDATE, I change this to 'yes' to track which records are updated before the constraint violation occurs. Using the FAIL resolution, these updates will remain unchanged, and I can track afterward how many records were updated.

```
CREATE TABLE test AS SELECT * FROM foods;
CREATE UNIQUE INDEX test_idx on test(id);
ALTER TABLE test ADD COLUMN modified text NOT NULL DEFAULT 'no';
SELECT COUNT(*) FROM test WHERE modified='no';

COUNT(*)
--------------------
412
```

```
UPDATE OR FAIL test SET id=800-id, modified='yes';
SQL error: column id is not unique

SELECT COUNT(*) FROM test WHERE modified='yes';

COUNT(*)
--------------------
387

DROP TABLE test;
```

There is one consideration with FAIL that you need to be aware of. The order that records are updated is nondeterministic. That is, you cannot be certain of the order of the records in the table or the order in which SQLite processes them. You might assume that it follows the order of the ROWID column, but this is not a safe assumption to make: there is nothing in the documentation that says so. The point is, if you are going to use FAIL, in many cases it might be better to use IGNORE. IGNORE will finish the job and modify all records that can be modified rather than bailing out on the first violation.

When defined within tables, conflict resolution is specified for individual columns. For example:

```
sqlite> CREATE TEMP TABLE cast(name text UNIQUE ON CONFLICT ROLLBACK);
sqlite> INSERT INTO cast VALUES ('Jerry');
sqlite> INSERT INTO cast VALUES ('Elaine');
sqlite> INSERT INTO cast VALUES ('Kramer');
```

The cast table has a single column name with a UNIQUE constraint and conflict resolution set to ROLLBACK. Any INSERT or UPDATE that triggers a constraint violation on name will be arbitrated by the ROLLBACK resolution rather than the default ABORT. The result will not only abort the statement but the entire transaction as well:

```
sqlite> BEGIN;
sqlite> INSERT INTO cast VALUES('Jerry');
SQL error: uniqueness constraint failed

sqlite> COMMIT;
SQL error: cannot commit - no transaction is active
```

COMMIT failed here because the name's conflict resolution already aborted the transaction. CREATE INDEX works the same way. Conflict resolution within tables and indices changes the default behavior of the operation from ABORT to that defined for the specific columns when those columns are the source of the constraint violation.

Conflict resolution at statement level (DML) overrides that defined at object level (DDL). Working from the previous example:

```
sqlite> BEGIN;
sqlite> INSERT OR REPLACE INTO cast VALUES('Jerry');
sqlite> COMMIT;
```

The REPLACE resolution in the INSERT overrides the ROLLBACK resolution defined on cast.name.

Database Locks

Locking is closely associated with transactions in SQLite. In order to use transactions effectively, you need to know a little something about how it does locking.

SQLite has coarse-grained locking. When a session is writing to the database, all other sessions are locked out until the writing session completes its transaction. To help with this, SQLite has a locking scheme that helps defer writer locks until the last possible moment in order to maximize concurrency.

SQLite uses a lock escalation policy whereby a connection gradually obtains exclusive access to a database in order to write to it. There are five different locking *states* in SQLite: *unlocked*, *shared*, *reserved*, *pending*, or *exclusive*. Each database session (or connection) can be in only one of these states at any given time. Furthermore, there is a corresponding lock for each state, except for unlocked—there is no lock required to be in the unlocked state.

To begin with, the most basic state is unlocked. In this state, no session is accessing data from the database. When you connect to a database, or even initiate a transaction with `BEGIN`, your connection is in the unlocked state.

The next state beyond unlocked is shared. In order for a session to read from the database (not write), it must first enter the shared state, and must therefore acquire a *shared lock*. Multiple sessions can simultaneously acquire and hold shared locks at any given time. Therefore, multiple sessions can read from a common database at any given time. However, no session can write to the database during this time—while any shared locks are active.

If a session wants to write to the database, it must first acquire a *reserved lock*. Only one reserved lock may be held at one time for a given database. Shared locks can coexist with a reserved lock. A reserved lock is the first phase of writing to a database. It does not block sessions with shared locks from reading, nor does it prevent sessions from acquiring new shared locks.

Once a session has a reserved lock, it can begin the process of making modifications; *however*, these modifications are cached and not actually written to disk. The reader's changes are stored in a memory cache (see the discussion of the `cache_size` pragma in the section "Database Configuration," later in this chapter, for more information).

When the session wants to commit the changes (or transaction) to the database, it begins the process of promoting its reserved lock to an *exclusive lock*. In order to get an exclusive lock, it must first promote its reserved lock to a *pending lock*. A pending lock starts a process of attrition whereby no new shared locks can be obtained. That is, other sessions with existing shared locks are allowed to continue as normal, but other sessions cannot acquire new shared locks. At this point, the session with the pending lock is waiting for the other sessions with shared locks to finish what they are doing and release them.

Once all of shared locks are released, the session with the pending lock can promote it to an exclusive lock. It is then free to make changes to the database. All of the previously cached changes are written to the database file.

Deadlocks

While you may find all of this interesting, you are probably wondering at this point why any of this matters. Why do you need to know this? Because if you don't know what you are doing, you can end up in a deadlock.

Consider the following scenario illustrated in Table 4-7. Two sessions, *A* and *B*—completely oblivious to one another—are working on the same database at the same time. Session A issues the first command, B the second and third, A the fourth, and so on.

Table 4-7. *A Portrait of a Deadlock*

Session A	Session B
`sqlite> BEGIN;`	
	`sqlite> BEGIN;`
	`sqlite> INSERT INTO foo VALUES ('x');`
`sqlite> SELECT * FROM foo;`	
	`sqlite> COMMIT;`
	`SQL error: database is locked`
`sqlite> INSERT INTO foo VALUES ('x');`	
`SQL error: database is locked`	

Both sessions wind up in a deadlock. Session *B* was the first to try to write to the database, and therefore has a pending lock. *A* attempts to write, but fails when `INSERT` tries to promote its shared lock to a reserved lock.

For the sake of argument, let's say that *A* decides to just wait around for the database to become writable. So does *B*. Then at this point, everyone else is effectively locked out too. If you try to open a third session, it won't even be able to read from the database. The reason is that *B* has a pending lock, which prevents any sessions from acquiring shared locks. So not only are *A* and *B* deadlocked, they have locked everyone else out of the database as well. Basically, you have a shared lock and one pending lock that don't want to relinquish control, and until one does, nobody can do anything.

So how do you avoid this? It's not like *A* and *B* can sit down in a meeting and work it out with their lawyers. *A* and *B* don't even know each other *exists*. The answer is to *pick the right transaction type for the job.*

Transaction Types

SQLite has three different transaction types that start transactions in different locking states. Transactions can be started as `DEFERRED`, `IMMEDIATE`, or `EXCLUSIVE`. A transaction's type is specified in the `BEGIN` command:

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

A *deferred transaction* does not acquire any locks until it has to. Thus with a deferred transaction, the `BEGIN` statement itself does nothing—it starts in the unlocked state. This is the default. If you simply issue a `BEGIN`, then your transaction is `DEFERRED`, and therefore sitting in the unlocked state. Multiple sessions can simultaneously start `DEFERRED` transactions at the same time without creating any locks. In this case, the first read operation against a database acquires a shared lock and similarly the first write operation *attempts* to acquire a reserved lock.

An *immediate transaction* attempts to obtain a reserved lock as soon as the BEGIN command is executed. If successful, BEGIN IMMEDIATE guarantees that no other session will be able to write to the database. As you know, other sessions can continue to read from the database, but the reserved lock prevents any new sessions from reading. Another consequence of the reserved lock is that no other sessions will be able to successfully issue a BEGIN IMMEDIATE or BEGIN EXCLUSIVE command. SQLite will return a SQLITE_BUSY error. During this time, you can make some modifications to the database, but you may not necessarily be able to commit them. When you call COMMIT, you could get SQLITE_BUSY. This means that there are other readers active, as in the earlier example. Once they are gone, you can commit the transaction.

An *exclusive transaction* obtains an exclusive lock on the database. This works similarly to IMMEDIATE, but when you successfully issue it, EXCLUSIVE guarantees that no other session is active in the database and that you can read or write with impunity.

The crux of the problem in the preceding example is that both sessions ultimately wanted to write to the database but they made no attempt to relinquish their locks. Ultimately, it was the shared lock that caused the problem. If both sessions had started with BEGIN IMMEDIATE, then the deadlock would not have occurred. In this case, only one of the sessions would have been able to enter BEGIN IMMEDIATE at one time, while the other would have to wait. The one that has to wait could keep retrying with the assurance that it would eventually get in. BEGIN IMMEDIATE and BEGIN EXCLUSIVE, if used by all sessions that want to write to the database, provide a synchronization mechanism, thereby preventing deadlocks. For this approach to work, though, everyone has to follow the rules.

The bottom line is this: if you are using a database that no other connections are using, then a simple BEGIN will suffice. If, however, you are using a database that other connections are also writing to, both you and they should use BEGIN IMMEDIATE or BEGIN EXCLUSIVE to initiate transactions. It works out best that way for both of you. Transactions and locks are covered in more detail in Chapter 5.

Database Administration

Database administration is generally concerned with controlling how a database operates. From a SQL perspective, this includes various topics such as views, triggers, and indexes. Additionally, SQLite includes some unique administrative features of its own, such the means to "attach" multiple databases to a single session, as well as database *pragmas*, which can be used for setting various configuration parameters.

Views

Views are virtual tables. They are also known as *derived tables*, as their contents are derived from other tables. While views look and feel like base tables, they aren't. The contents of base tables are persistent whereas the contents of views are dynamically generated when they are used. Specifically, a view is composed of relational expressions that take other tables and produce a new table. The syntax to create a view is as follows:

```
CREATE VIEW name AS sql;
```

The name of the view is given by *name* and its definition by *sql*. The resulting view will look like a base table named name. Imagine you had a query you ran all the time, so much that you get sick of writing it. Views are the cure for this particular sickness. Say your query was as follows:

```
SELECT f.name, ft.name, e.name
FROM foods f
INNER JOIN food_types ft on f.type_id=ft.id
INNER JOIN foods_episodes fe ON f.id=fe.food_id
INNER JOIN episodes e ON fe.episode_id=e.id;
```

This returns the name of every food, its type, and every episode it was in. It is one big table of 504 rows with just about every food fact. Rather than having to write out (or remember) the previous query every time you want these results, you can tidily restate it in the form of a view. Let's name it `details`:

```
CREATE VIEW details AS
SELECT f.name AS fd, ft.name AS tp, e.name AS ep, e.season as ssn
FROM foods f
INNER JOIN food_types ft on f.type_id=ft.id
INNER JOIN foods_episodes fe ON f.id=fe.food_id
INNER JOIN episodes e ON fe.episode_id=e.id;
```

Now you can query `details` just as you would a table. For example:

```
sqlite> SELECT fd as Food, ep as Episode
        FROM details WHERE ssn=7 AND tp like 'Drinks';

Food                  Episode
--------------------  --------------------
Apple Cider           The Bottle Deposit 1
Bosco                 The Secret Code
Cafe Latte            The Postponement
Cafe Latte            The Maestro
Champagne Coolies     The Wig Master
Cider                 The Bottle Deposit 2
Hershey's             The Secret Code
Hot Coffee            The Maestro
Latte                 The Maestro
Mellow Yellow soda    The Bottle Deposit 1
Merlot                The Rye
Orange Juice          The Wink
Tea                   The Hot Tub
Wild Turkey           The Hot Tub
```

The contents of views are dynamically generated. Thus, every time you use `details`, its associated SQL will be reexecuted, producing results based on the data in the database at that moment.

Views also have other purposes, such as security, although this particular kind of security does not exist in SQLite. Some databases offer row- and column-level security in which only specific users, groups, or roles can view or modify specific parts of tables. In such databases, views can be defined on tables to exclude sensitive columns, allowing users with less security privileges to access parts of tables that are not secured. For example, say you have a table `secure`, defined as follows:

```
CREATE TABLE secure (id int, public_data text, restricted_data text);
```

You may want to allow users access to the first two columns but not the third. In other databases, you would limit access to `secure` to just the users who can access all columns. You would then create a view that contains just the first two columns that everyone else can look at:

```
CREATE VIEW public_secure AS SELECT id, public_data FROM secure;
```

Some of this kind of security is available if you program with SQLite, using its operational control facilities. This is covered in Chapters 5 and 6.

Finally, to drop a view, use the `DROP VIEW` command:

```
DROP VIEW name;
```

The name of the view to drop is given by `name`.

Materialized Views

The relational model calls for updatable views, sometimes referred to as *materialized views*. These are views that you can modify. You can run `INSERT` or `UPDATE` statements on them, for example, and the respective changes are applied directly to their underlying tables. Materialized views are not supported in SQLite. However, using triggers, you can create something that looks like materialized views. These are covered in the section "Triggers."

Indexes

Indexes are a construct designed to speed up queries under certain conditions. Consider the following query:

```
SELECT * FROM foods WHERE name='JujyFruit';
```

When a database searches for matching rows, the default method it uses to perform this is called a *sequential scan*. That is, it literally searches (or scans) every row in the table to see if its `name` attribute matches `'JujyFruit'`.

However, if this query is used frequently, and `foods` was very large, there is another method available that can be much faster, called an *index scan*. An index is a special disk-based data structure (called a B-tree), which stores the values for an entire column (or columns) in a highly organized way that is optimized for searching.

The search speed of these two methods can be represented mathematically. The search speed of a sequential scan is proportional to the number of rows in the table: the more rows, the longer the scan. This relationship—bigger table, longer search—is called *linear time*, as in "the search method operates in linear time." It is represented mathematically using what is called the *Big O notation*, which in this case is $O(n)$, where n stands for the number of elements (or rows) in the set. The index scan, on the other hand, has $O(log(n))$ search time, or logarithmic time. This is much faster. If you have a table of 10,000 records, a sequential scan will read all 10,000 rows to find all matches, while an index scan will read 4 rows ($log(10{,}000)$) to find the first match (and from that point on it would be linear time to find all subsequent matches). This is quite a speed-up.

But there is no such thing as a free lunch. Indexes also increase the size of the database. They literally keep a copy of all columns they index. If you index every column in a table, you effectively double the size of the table. Another consideration is that indexes must be maintained. When you insert, update, or delete records, in addition to modifying the table, the database must modify each and every index on that table as well. So indexes can slow down inserts, updates, and similar operations.

But in moderation, indexes can make a huge performance difference. Whether or not to add an index is more subjective than anything else. Different people have different criteria for what is acceptable. Indexes illustrate one of the great things about SQL: you only need to specify what to get and not how to get it. Because of this, you can often optimize your database (such as by adding or removing indexes) without having to rewrite your code. Just create an index in the right place.

The command to create an index is as follows:

```
CREATE INDEX [UNIQUE] index_name ON table_name (columns)
```

The variable `index_name` is the name of the index, and must be unique, and `table_name` is the name of the table containing the column(s) to index. The variable `columns` is either a single column or a comma-separated list of columns.

If you use the `UNIQUE` keyword, then the index will have the added constraint that all values in the index must be unique. This applies to both the index, and by extension, to the column or columns it indexes. The `UNIQUE` constraint covers all columns defined in the index, and it is their combined values (not individual values) that must be unique. For example:

```
sqlite> CREATE TABLE foo(a text, b text);
sqlite> CREATE UNIQUE INDEX foo_idx on foo(a,b);
sqlite> INSERT INTO foo VALUES ('unique', 'value');
sqlite> INSERT INTO foo VALUES ('unique', 'value2');
sqlite> INSERT INTO foo VALUES ('unique', 'value');
SQL error: columns a, b are not unique
```

You can see here that uniqueness is defined by both columns collectively, not individually. Notice that collation plays an important role here as well.

To remove an index, use the `DROP INDEX` command, which is defined as follows:

```
DROP INDEX index_name;
```

Collation

Each column in the index can have a collation associated with it. For example, to create a case-insensitive index on `foods.name`, you'd use the following:

```
CREATE INDEX foods_name_idx on foods (name COLLATE NOCASE);
```

This means that values in the `name` column will sort without respect to case. You can list the indexes for a table in the SQLite command-line program by using the `.indices` shell command. For example:

```
sqlite> .indices foods
foods_name_idx
For more information, you can use the .schema shell command as well:
sqlite> .schema foods
CREATE TABLE foods(
  id integer primary key,
  type_id integer,
  name text );
CREATE INDEX foods_name_idx on foods (name COLLATE NOCASE);
```

You can also obtain this information by querying the `sqlite_master` table, described later in this section.

Index Utilization

It is important to understand when indexes are used and when they aren't. There are very specific conditions in which SQLite will decide to use an index. SQLite will use a single column index, if available, for the following expressions in the `WHERE` clause:

```
column {=|>|>=|<=|<} expression
expression {=|>|>=|<=|<} column
column IN (expression-list)
column IN (subquery)
```

Multicolumn indexes have more specific conditions before they are used. This is perhaps best illustrated by example. Say you have a table defined as follows:

```
CREATE TABLE foo (a,b,c,d);
```

Furthermore, you create a multicolumn index as follows:

```
CREATE INDEX foo_idx on foo (a,b,c,d);
```

The columns of `foo_idx` can only be used sequentially from left to right. That is, in the query

```
SELECT * FROM foo WHERE a=1 AND b=2 AND d=3
```

only the first and second conditions will use the index. The reason the third condition was excluded is because there was no condition that used `c` to bridge the gap to `d`. Basically, when SQLite uses a multicolumn index, it works from left to right column-wise. It starts with the left column and looks for a condition using that column. It moves to the second column, and so on. It continues until either it fails to find a valid condition in the `WHERE` clause that uses it or there are no more columns in the index to use.

But there is one more requirement. SQLite will use a multicolumn index only if all of the conditions use either the equality (`=`) or `IN` operator for all index columns *except for the right-most index column*. For that column, you can specify up to two inequalities to define its upper and lower bounds. Consider, for example:

```
SELECT * FROM foo WHERE a>1 AND b=2 AND c=3 AND d=4
```

SQLite will only do an index scan on column a. The a>1 expression becomes the rightmost index column because it uses the inequality. All columns after it are not eligible to be used as a result. Similarly,

```
SELECT * FROM foo WHERE a=1 AND b>2 AND c=3 AND d=4
```

will use the index columns a and b and stop there as b>2 becomes the rightmost index term by its use of an inequality operator.

An off-the-cuff way to time a query within a SQL statement is to use a subselect in the FROM clause returning the current time, which will be joined to your input relation(s). Then select the current time in the SELECT clause. The time in the FROM clause will be computed at the start of the query. The time in the SELECT clause will be computed as each row is processed. Therefore, the difference between the two times in the last row of the result set will be your relative query time. For example, the following SQL is a triple Cartesian join on food_types. It's quite slow, as it should be. The results display the last five records of the result set.

```
SELECT CAST(strftime('%s','now') as INTEGER)-CAST(time.start as INTEGER) time,
       ft1.id, ft2.id, ft3.id, ft4.id
FROM food_types ft1, food_types ft2, food_types ft3, food_types ft4,
     (SELECT strftime('%s','now') start) time;
```

```
18    15    15    15    11
18    15    15    15    12
18    15    15    15    13
18    15    15    15    14
18    15    15    15    15
```

The first column is the elapsed time in seconds. This query took 18 seconds. Although this doesn't represent the actual query time because there is timing overhead, relative query time is all you need to judge an index's worth. If this were an actual slow query that you were trying to optimize, you would now add the index you think might help and rerun the query. Is it significantly faster? If so, then you may have a useful index.

In short, when you create an index, have a reason for creating it. Make sure there is a specific performance gain you are getting before you take on the overhead that comes with it. Well-chosen indexes are a wonderful thing. Indexes that are thoughtlessly scattered here and there in the vain hope of performance are of dubious value.

Triggers

Triggers execute specific SQL commands when specific database events transpire on specific tables. The general syntax for creating a trigger is as follows:

```
CREATE [TEMP|TEMPORARY] TRIGGER name [BEFORE|AFTER]
  [INSERT|DELETE|UPDATE|UPDATE OF columns] ON table
  action
```

A trigger is defined by a name, an action, and a table. The action, or trigger body, consists of a series of SQL commands. Triggers are said to *fire* when such events take place. Furthermore, triggers can be made to fire before or after the event using the BEFORE or AFTER keyword, respectively. Events include DELETE, INSERT, and UPDATE commands issued on the specified table. Triggers can be used to create your own integrity constraints, log changes, update other tables, and many other things. They are limited only by what you can write in SQL.

UPDATE Triggers

UPDATE triggers, unlike INSERT and DELETE triggers, may be defined for specific columns in a table. The general form of this kind of trigger is as follows:

```
CREATE TRIGGER name [BEFORE|AFTER] UPDATE OF column ON table
action
```

The following is a SQL script that shows an UPDATE trigger in action:

```
.h on
.m col
.w 50
.echo on
CREATE TEMP TABLE log(x);

CREATE TEMP TRIGGER foods_update_log UPDATE of name ON foods
BEGIN
  INSERT INTO log VALUES('updated foods: new name=' || NEW.name);
END;

BEGIN;
UPDATE foods set name='JUJYFRUIT' where name='JujyFruit';
SELECT * FROM log;
ROLLBACK;
```

This script creates a temporary table called log, as well as a temporary UPDATE trigger on foods.name that inserts a message into log when it fires. The action takes place inside the transaction that follows. The first step of the transaction updates the name column of the row whose name is 'JUJYFRUIT'. This causes the UPDATE trigger to fire. When it fires, it inserts a record into the log. Next, the transaction reads log, which shows that the trigger did indeed fire. The transaction then rolls back the change, and when the session ends, the log table and the UPDATE trigger are destroyed. Running the script produces the following output:

```
mike@linux tmp # sqlite3 foods.db < trigger.sql
```

```
CREATE TEMP TABLE log(x);

CREATE TEMP TRIGGER foods_update_log AFTER UPDATE of name ON foods
BEGIN
  INSERT INTO log VALUES('updated foods: new name=' || NEW.name);
END;
```

```
BEGIN;
UPDATE foods set name='JUJYFRUIT' where name='JujyFruit';
SELECT * FROM log;
x
--------------------------------------------------
updated foods: new name=JUJYFRUIT
ROLLBACK;
```

SQLite provides access to both the old (original) row and the new (updated) row in `UPDATE` triggers. The old row is referred to as `OLD` and the new row as `NEW`. Notice in the script how the trigger refers to `NEW.name`. All attributes of both rows are available in `OLD` and `NEW` using the dot notation. I could have just as easily recorded `NEW.type_id` or `OLD.id`.

Error Handling

Defining a trigger before an event takes place gives you the opportunity to stop the event from happening. `BEFORE` triggers enable you to implement new integrity constraints. SQLite provides a special SQL function for triggers called `RAISE()`, which allows them to raise an error within the trigger body. `RAISE` is defined as follows:

RAISE(*resolution*, *error_message*);

The first argument is a conflict resolution policy (`ABORT`, `FAIL`, `IGNORE`, `ROLLBACK`, etc.). The second argument is an error message. If you use `IGNORE`, the remainder of the current trigger along with the SQL statement that caused the trigger to fire, as well as any subsequent triggers that would have been fired, are all terminated. If the SQL statement that caused the trigger to fire is itself part of another trigger, then that trigger resumes execution at the beginning of the next SQL command in the trigger action.

Conflict Resolution

If a conflict resolution policy is defined for a SQL statement that causes a trigger to fire, then that policy supersedes the policy defined within the trigger. If, on the other hand, the SQL statement does not have any conflict resolution policy defined, and the trigger does, then the trigger's policy is used.

Updatable Views

Triggers make it possible to create something like materialized views, as mentioned earlier in this chapter. In reality, they aren't true materialized views but rather more like updatable views. With true materialized views, the view is updatable all by itself—you define the view and it figures out how to map all changes to its underlying base tables. This is not a simple thing. Nor is it supported in SQLite. However, using triggers, we can create the appearance of a materialized view.

The idea here is that you create a view and then create a trigger that handles update events on that view. SQLite supports triggers on views using the `INSTEAD OF` keywords in the trigger definition. To illustrate this, let's create a view that combines `foods` with `food_types`:

```
CREATE VIEW foods_view AS
  SELECT f.id fid, f.name fname, t.id tid, t.name tname
  FROM foods f, food_types t;
```

This view joins the two tables according to their foreign key relationship. Notice that I have created aliases for all column names in the view. This allows me to differentiate the respective `id` and `name` columns in each table when I reference them from inside the trigger. Now, let's make the view updatable by creating an `UPDATE` trigger on it:

```
CREATE TRIGGER on_update_foods_view
INSTEAD OF UPDATE ON foods_view
FOR EACH ROW
BEGIN
   UPDATE foods SET name=NEW.fname WHERE id=NEW.fid;
   UPDATE food_types SET name=NEW.tname WHERE id=NEW.tid;
END;
```

Now if you try to update the `foods_view`, this trigger gets called. The trigger simply takes the values provided in the `UPDATE` statement and uses them to update the underlying base tables `foods` and `food_types`. Testing it out yields the following:

```
.echo on
-- Update the view within a transaction
BEGIN;
UPDATE foods_view SET fname='Whataburger', tname='Fast Food' WHERE fid=413;
-- Now view the underlying rows in the base tables:
SELECT * FROM foods f, food_types t WHERE f.type_id=t.id AND f.id=413;
-- Roll it back
ROLLBACK;
-- Now look at the original record:
SELECT * FROM foods f, food_types t WHERE f.type_id=t.id AND f.id=413;
```

```
BEGIN;
UPDATE foods_view SET fname='Whataburger', tname='Fast Food' WHERE fid=413;
SELECT * FROM foods f, food_types t WHERE f.type_id=t.id AND f.id=413;
id   type_id  name            id   name
---  -------  --------------  ---  ---------
413  1        Whataburger     1    Fast Food

ROLLBACK;

SELECT * FROM foods f, food_types t WHERE f.type_id=t.id AND f.id=413;
id   type_id  name            id   name
---  -------  --------------  ---  -------
413  1        Cinnamon Bobka  1    Bakery
```

You can just as easily add `INSERT` and `DELETE` triggers and have the rough equivalent of a materialized view.

Foreign Key Constraints Using Triggers

One of the most interesting applications of triggers in SQLite I have seen is the implementation of foreign key constraints, originally posted on the SQLite Wiki (www.sqlite.org/contrib). To further explore triggers, I will use this idea to implement foreign key constraints between foods and food_types.

As mentioned earlier, foods.type_id references food_types.id. Therefore, every value in foods.type_id should correspond to some value in food_types.id. The first step in enforcing this constraint lies in controlling what can be inserted into foods. This is accomplished with the following INSERT trigger:

```
CREATE TRIGGER foods_insert_trg
BEFORE INSERT ON foods
BEGIN
  SELECT CASE
     WHEN (SELECT id FROM food_types WHERE id=NEW.type_id) IS NULL
     THEN RAISE( ABORT,
     'Foreign Key Violation: foods.type_id is not in food_types.id')
  END;
END;
```

This trigger runs a subquery that checks for the value of NEW.type_id in foods_types.id. If no match is found, the subquery returns NULL, which triggers the WHEN condition, calling the RAISE function.

After installing the trigger, the following SQL tries to insert a record with an invalid type_id (the maximum id value in food_types is 15):

```
sqlite> INSERT INTO foods VALUES (NULL, 20, 'Blue Bell Ice Cream');
SQL error: Foreign Key Violation: foods.type_id is not in food_types.id
```

Next is UPDATE. The only thing that matters on UPDATE of foods is the type_id field, so the trigger will be defined on that column alone. Aside from this and the trigger's name, the trigger is identical to INSERT:

```
CREATE TRIGGER foods_update_trg
BEFORE UPDATE OF type_id ON foods
BEGIN
  SELECT CASE
     WHEN (SELECT id FROM food_types WHERE id=NEW.type_id) IS NULL
     THEN RAISE(ABORT,
    'Foreign Key Violation: foods.type_id is not in food_types.id')
  END;
END;
```

Testing this trigger reveals the same results:

```
sqlite> UPDATE foods SET type_id=20 WHERE name='Chocolate Bobka';
SQL error: Foreign Key Violation: foods.type_id is not in food_types.id
```

The final piece of the puzzle is DELETE. Deleting rows in foods doesn't affect the relationship with food_types. Deleting rows in food_types, however, does affect the relationship with foods. If a row is deleted in food_types, then there could potentially be rows in foods that may reference it, in which case the relationship has been compromised. Therefore, we need a DELETE trigger on food_types that does not allow the deletion of a row if there are rows in foods that reference it. To that end, the DELETE trigger is defined as follows:

```
CREATE TRIGGER foods_delete_trg
BEFORE DELETE ON food_types
BEGIN
  SELECT CASE
     WHEN (SELECT COUNT(type_id) FROM foods WHERE type_id=OLD.id) > 0
     THEN RAISE(ABORT,
     'Foreign Key Violation: foods rows reference row to be deleted.')
  END;
END;
```

After installing this trigger, if I try to delete the 'Bakery' row in food_types I get:

```
sqlite> DELETE FROM food_types WHERE name='Bakery';
SQL error: Foreign Key Violation: foods rows reference row to be deleted.
```

To make sure this works under the correct conditions:

```
sqlite> BEGIN;
sqlite> DELETE FROM foods WHERE type_id=1;
sqlite> DELETE FROM food_types WHERE name='Bakery';
sqlite> ROLLBACK;
```

The DELETE trigger allows the delete if there are no rows in foods referencing it.

So there you have it: simple, trigger-based foreign key constraints. As mentioned earlier, while SQLite does support CHECK constraints, triggers can pretty much do everything CHECK constraints can and then some.

Attaching Databases

SQLite allows you to "attach" multiple databases to the current session using the ATTACH command. When you attach a database, all of its contents are accessible in the global scope of the current database file. ATTACH has the following syntax:

ATTACH [DATABASE] *filename* AS *database_name*;

Here, filename refers to the path and name of the SQLite database file, and database_name refers to the logical name with which to reference that database and its objects. The main database is automatically assigned the name main. If you create any temporary objects, then SQLite will create an attached database name temp. (You can see these objects using the database_list pragma, described later.) The logical name may be used to reference objects within the attached database. If there are tables or other database objects that share the same name in both databases, then the logical name is required to reference such objects in the attached database. For example, if both databases have a table called foo, and the logical name of the attached database is db2, then the only way to query foo in db2 is by using the fully qualified name foo.db2, as follows:

```
sqlite> ATTACH DATABASE '/tmp/db' as db2;
sqlite> SELECT * FROM db2.foo;

x
----------
bar
```

If you really want to, you can qualify objects in the main database using the name `main`:

```
sqlite> SELECT * FROM main.foods LIMIT 2;

id          type_id     name
----------  ----------  --------------
1           1           Bagels
2           1           Bagels, raisin
```

The same is true with the temporary database:

```
sqlite> CREATE TEMP TABLE foo AS SELECT * FROM food_types LIMIT 3;
sqlite> SELECT * FROM temp.foo;

id   name
---  -------------
1    Bakery
2    Cereal
3    Chicken/Fowl
```

You detach databases with the `DETACH DATABASE` command, defined as follows:

```
DETACH [DATABASE] database_name;
```

This command takes the logical name of the attached database (given by *database_name*) and detaches the associated database file. You get a list of attached databases using the `database_list` pragma, explained in the section "Database Configuration."

Cleaning Databases

SQLite has two commands designed for cleaning—`REINDEX` and `VACUUM`. `REINDEX` is used to rebuild indexes. It has two forms:

```
REINDEX collation_name;
REINDEX table_name|index_name;
```

The first form rebuilds all indexes that use the collation name given by `collation_name`. It is only needed when you change the behavior of a user-defined collating sequence (e.g., multiple sort orders in Chinese). All indexes in a table (or a particular index given its name) can be rebuilt with the second form.

`VACUUM` cleans out any unused space in the database by rebuilding the database file. `VACUUM` will not work if there are any open transactions. An alternative to manually running `VACUUM` statements is autovacuum. This feature is enabled using the `auto_vacuum` pragma, described in the next section.

Database Configuration

SQLite doesn't have a configuration file. Rather, all of its configuration parameters are implemented using *pragmas*. Pragmas work in different ways. Some are like variables; others are like commands. They cover many aspects of the database, such as runtime information, database schema, versioning, file format, memory use, and debugging. Some pragmas are read and set like variables, while others require arguments and are called like functions. Many pragmas have both temporary and permanent forms. Temporary forms affect only the current session for the duration of its lifetime. The permanent forms are stored in the database and affect every session. The cache size is one such example.

This section covers the most commonly used pragmas. A complete list of all SQLite pragmas can be found in Appendix A.

The Connection Cache Size

The cache size pragmas control how many database pages a session can hold in memory. To set the default cache size for the current session, you use the `cache_size` pragma:

```
sqlite> PRAGMA cache_size;

cache_size
---------------
2000

sqlite> PRAGMA cache_size=10000;
sqlite> PRAGMA cache_size;

cache_size
---------------
10000
```

You can permanently set the cache size for all sessions using the `default_cache_size` pragma. This setting is stored in the database. This will only take effect for sessions created after the change, not for currently active sessions.

One of the uses for the cache is in storing pending changes when a session is in a `RESERVED` state (it has a `RESERVED` lock), as described earlier in the section "Transactions." If the session fills up the cache, it will not be able to continue further modifications until it gets an `EXCLUSIVE` lock, which means that it may have to first wait for readers to clear.

If you or your program(s) perform many updates or deletes on a database that is being used by many other sessions, it may help you to increase the cache size. The larger the cache size, the more modifications a session can cache change before it has to get an `EXCLUSIVE` lock. This not only allows a session to get more work done before having to wait, it also cuts down on the time the exclusive locks needs to be held, as all the work is done up front. In this case, the `EXCLUSIVE` lock only needs to be held long enough to flush the changes in the cache to disk. Some tips for tuning the cache size are covered in Chapter 5.

Getting Database Information

You can obtain database information using the database schema pragmas, defined as follows:

- `database_list`: Lists information about all attached databases.
- `index_info`: Lists information about the columns within an index. It takes an index name as an argument.
- `index_list`: Lists information about the indexes in a table. It takes a table name as an argument.
- `table_info`: Lists information about all columns in a table.

The following illustrates some information provided by these pragmas:

```
sqlite> PRAGMA database_list;

seq   name     file
----  -------  --------------------
0     main     /tmp/foods.db
2     db2      /tmp/db

sqlite> CREATE INDEX foods_name_type_idx ON foods(name,type_id);
sqlite> PRAGMA index_info(foods_name_type_idx);

seqn  cid      name
----  -------  --------------------
0     2        name
1     1        type_id

sqlite> PRAGMA index_list(foods);

seq    name                unique
-----  ------------------- ---------------
0      foods_name_type_idx 0

sqlite> PRAGMA table_info(foods);

cid    name             type             notn  dflt  pk
-----  ---------------  ---------------  ----  ----  ----------
0      id               integer          0           1
1      type_id          integer          0           0
2      name             text             0           0
```

Synchronous Writes

Normally, SQLite commits all changes to disk at critical moments to ensure transaction durability. However, it is possible to turn this off for performance gains. You do this with the `synchronous` pragma. There are three settings: `FULL`, `NORMAL`, and `OFF`. They are defined as follows:

- FULL: SQLite will pause at critical moments to make sure that data has actually been written to the disk surface before continuing. This ensures that if the operating system crashes or if there is a power failure, the database will be uncorrupted after rebooting. FULL synchronous is very safe, but it is also slow.
- NORMAL: SQLite will still pause at the most critical moments, but less often than in FULL mode. There is a very small (though non-zero) chance that a power failure at just the wrong time could corrupt the database in NORMAL mode. But in practice, you are more likely to suffer a catastrophic disk failure or some other unrecoverable hardware fault.
- OFF: SQLite continues operation without pausing as soon as it has handed data off to the operating system. This can speed up some operations as much as 50 or more times. If the application running SQLite crashes, the data will be safe. However, if the operating system crashes or the computer loses power, the database may be corrupted.

There is no persistent form of the synchronous pragma. Chapter 5 explains this setting's crucial role in transaction durability and how it works.

Temporary Storage

Temporary storage is where SQLite keeps transient data such as temporary tables, indexes, and other objects. By default, SQLite uses a compiled-in location, which varies between platforms. There are two pragmas that govern temporary storage: temp_store and temp_store_directory. The first pragma determines whether SQLite uses memory or disk for temporary storage. There are actually three possible values: DEFAULT, FILE, or MEMORY. DEFAULT uses the compiled-in default, FILE uses an operating system file, and MEMORY uses RAM. If FILE is set as the storage medium, then the second pragma, temp_store_directory, can be used to set the directory in which the temporary storage file is placed.

Page Size, Encoding, and Autovacuum

The database page size, encoding, and autovacuuming must be set before a database is created. That is, in order to alter the defaults, you must first set these pragmas before creating any database objects in a new database. The defaults are a 1,024-byte page size and UTF-8 encoding. SQLite supports page sizes ranging from 512 to 32,786 bytes, in powers of 2. Supported encodings are UTF-8, UTF-16le (little-endian UTF-16 encoding), and UTF-16be (big-endian UTF-16 encoding).

A database's size can be automatically kept to a minimum using the auto_vacuum pragma. Normally, when a transaction that deletes data from a database is committed, the database file remains the same size. When the auto_vacuum pragma is enabled, the database file shrinks when a transaction that deletes data is committed. To support this functionality, the database stores extra information internally, resulting in slightly larger database files than would otherwise be possible. The VACUUM command has no effect on databases that use auto_vacuum.

Debugging

There are four pragmas for various debugging purposes. The integrity_check pragma looks for out-of-order records, missing pages, malformed records, and corrupted indexes. If any problems are found, then a single string is returned describing the problems. If everything is in order, SQLite returns ok. The other pragmas are used for tracing the parser and virtual database engine and can only be enabled if SQLite is compiled with debugging information. Detailed information on these pragmas can be found in Chapter 9.

The System Catalog

The sqlite_master table is a system table that contains information about all the tables, views, indexes, and triggers in the database. For example, the current contents of the foods database are as follows:

```
sqlite> SELECT type, name, rootpage FROM sqlite_master;

type        name                       rootpage
----------  -------------------------  ----------
table       episodes                   2
table       foods                      3
table       foods_episodes             4
table       food_types                 5
index       foods_name_idx             30
table       sqlite_sequence            50
trigger     foods_update_trg           0
trigger     foods_insert_trg           0
trigger     foods_delete_trg           0
```

The type column refers to the type of object, name is of course the name of the object, and rootpage refers to the first B-tree page of the object in the database file. This latter column is only relevant for tables and indexes.

The sqlite_master table also contains another column called sql, which stores the DML used to create the object. For example:

```
sqlite> SELECT sql FROM sqlite_master WHERE name='foods_update_trg';

CREATE TRIGGER foods_update_trg
BEFORE UPDATE OF type_id ON foods
BEGIN
  SELECT CASE
     WHEN (SELECT id FROM food_types WHERE id=NEW.type_id) IS NULL
     THEN RAISE( ABORT,
                 'Foreign Key Violation: foods.type_id is not in food_types.id')
  END;
END
```

Viewing Query Plans

You can view the way SQLite goes about executing a query by using the EXPLAIN command. EXPLAIN lists the VDBE program that would be used to carry out a SQL command. The VDBE is the virtual machine in SQLite that carries out all of its database operations. Every query in SQLite is executed by first compiling the SQL into VDBE byte code and then running the byte code. For example:

```
sqlite> EXPLAIN SELECT * FROM foods;

addr  opcode           p1    p2    p3
----  ---------------  ----  ----  ----
0     Goto             0     12
1     Integer          0     0
2     OpenRead         0     3
3     SetNumColumns    0     3
4     Rewind           0     10
5     Rowid            0     0
6     Column           0     1
7     Column           0     2
8     Callback         3     0
9     Next             0     5
10    Close            0     0
11    Halt             0     0
12    Transaction      0     0
13    VerifyCookie     0     134
14    Goto             0     1
15    Noop             0     0
```

Studying these query plans is not for the faint of heart. The average person is not going to find a VDBE program very intuitive. However, for those who are willing to try, the VDBE, its op codes, and theory of operation are covered in Chapter 9.

Summary

SQL may be a simple language to use, but there is quite a bit of it. But that shouldn't be too surprising, as it is the sole interface through which to interact with a relational database. Whether you are a casual user, system administrator, or developer, you have to know SQL if you are going to work with a relational database.

SQL is a thin wrapper around the relational model. It is composed of three essential parts: form, function, and integrity. Form relates to how information is represented. In SQL, this is done with the table. The table is the sole unit of information, and is composed of rows and columns. The database is made up of other objects as well, such as views, indexes, and triggers. Views provide a convenient way to represent information in tables in different forms. Indexes work to speed up queries. Triggers enable you to associate programming logic with specific database events as they transpire on tables. All of these objects are created using data definition language (DDL). DDL is the part of SQL reserved for creating and destroying database objects. Furthermore, all objects in a database are listed in the `sqlite_master` system table.

The functional part of SQL pertains to how information is manipulated. The part of SQL reserved for this is called data manipulation language (DML). DML is composed of the `SELECT`, `UPDATE`, `INSERT`, and `DELETE` commands. `SELECT` is the largest and most complicated command in the language. `SELECT` employs 12 of the 14 operations defined in relational algebra. It arranges those operations in a customizable pipeline. You can use those operations as needed to combine, filter, order, and aggregate information within tables.

The integrity part of SQL controls how data is inserted and modified. It protects information and relationships by ensuring that the values in columns conform to specific rules. It is implemented using constraints, such as `UNIQUE`, `CHECK`, and `NOT NULL`.

SQLite has a unique way of handling data types, unlike most other relational databases. All values have a storage class. SQLite has five storage classes: `INTEGER`, `REAL`, `TEXT`, `BLOB`, and `NULL`. These classes specify how values are stored and manipulated in the database. Furthermore, all columns have an affinity, which determines how values are stored within them. SQLite has four kinds of affinity: `NUMERIC`, `INTEGER`, `TEXT`, and `NONE`. When SQLite stores a value in a column, it uses both the column's affinity and the value's storage class to determine how to store the value. For those who have to have strict type checking on a column, it can be implemented using the `typeof()` function and a `CHECK` constraint.

If you are programming with SQLite, then you should be off to a good start on the SQL side of things. Now you need to know a little about how SQLite goes about executing all of these commands. This is where Chapter 5 should prove useful. It will introduce you to the API and show you how it works in relation to the way SQLite functions internally.