Using

SQLite

O'REILLY®

Jay A. Kreibich

Table of Contents

Chapter 6. Database Design. Using SQLite, ISBN: 9781449394592

CHAPTER 6

Database Design

Relational databases have only one type of data container: the table. When designing any database, the main concern is defining the tables that make up the database and defining how those tables reference and interact with each other.

Designing tables for a database is a lot like designing classes or data structures for an application. Simple designs are largely driven by common sense, but as things get larger and more complex, it takes some experience and insight to keep the design clean and on target. Understanding basic design principles and standard approaches can be a big help.

Tables and Keys

Tables may look like simple two-dimensional grids of simple values, but a well-defined table has a fair amount of structure. Different columns can play different roles. Some columns act as unique identifiers that define the intent and purpose of each row. Other columns hold supporting data. Still other columns act as external references that link rows in one table to rows in another table. When designing a table, it is important to understand why each column is there, and what role each column is playing.

Keys Define the Table

When designing a table, you usually start by specifying the *primary key*. The primary key consists of one or more columns that uniquely identify each row of a table. In a sense, the primary key values represent the fundamental identity of each row in the table. The primary key columns identify what the table is all about. All the other columns should provide supporting data that is directly relevant to the primary key.

Sometimes the primary key is an actual unique data value, such as a room number or a hostname. Very often the primary key is simply an arbitrary identification number, such as an employee or student ID number. The important point is that primary keys must be unique over every *possible* entry in the table, not just the rows that happen to be in there right now. This is why names are normally not used as primary keys—in a large group of people, it is too easy to end up with duplicate (or very similar) names.

While there is nothing particularly special about the columns that make up a primary key, the keys themselves play a very important role in the design and use of a database. Their role as a unique identifier for each row makes them analogous to the key of a hash table, or the key to a dictionary class of data structure. They are essentially "lookup" values for the rows of a table.

A primary key can be identified in the `CREATE TABLE` command. Explicitly identifying the primary key will cause the database system to automatically create a `UNIQUE` index across all of the primary key columns. Declaring the primary key also allows for some syntax shortcuts when establishing relationships between tables.

For example, this table definition defines the `employee_id` field to be a primary key:

```
CREATE TABLE employee (
    employee_id   INTEGER   PRIMARY KEY   NOT NULL,
    name          TEXT   NOT NULL
    /* ...etc... */
);
```

For more information on the syntax used to define a primary key, see the section "Primary keys" on page 40.

In schema documentation, primary keys are often indicated with the abbreviation "PK." It is also common to double underline primary keys when drawing out tables, as shown in Figure 6-1.

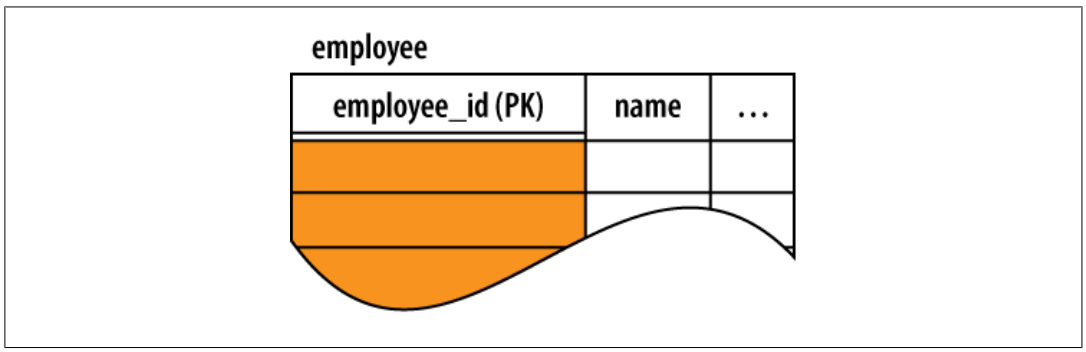

Figure 6-1. Primary keys are sometimes identified with the abbreviation PK. It is also common to use a double underline when diagramming the table.

Many database queries use a table's primary key as either the input or output. The database might be asked to return the row with a given key, such as, "return the record for employee #953." It is also common to ask for collections of keys, such as, "gather the ID values for all employees hired more than two years ago." This set of keys might then be joined to another table as part of a report.

Foreign Keys

In addition to identifying each row in a table, primary keys are also central to joining tables together. Since the primary key acts as a unique row identifier, you can create a reference to a specific row by recording the row's primary key. This is known as a *foreign key*. A foreign key is a copy or recording of a primary key, and is used as a reference or pointer to a different (or "foreign") row, most often in a different table.

Like the primary key, columns that make up a foreign key can be identified within the `CREATE TABLE` command. In this example, we define the format of a task assignment. Each task gets assigned to a specific employee by referencing the `employee_id` field from the `employee` table:

```
CREATE TABLE task_assignment (
    task_assign_id   INTEGER   PRIMARY KEY,
    task_name        TEXT      NOT NULL,
    employee_id      INTEGER   NOT NULL   REFERENCES employee( employee_id )
    /* ...etc... */
);
```

The `REFERENCES` constraint indicates that this column is a foreign key. The constraint indicates which table is referenced and, optionally, which column. If no column is indicated, the foreign key will reference the primary key (meaning the column reference used in the prior example is not required, since `employee_id` is the primary key of the `employee` table). The vast majority of foreign keys will reference a primary key, but if a column other than the primary key is used, that column must have a `UNIQUE` constraint, or it must have a single-column `UNIQUE` index.

A foreign key can also be defined as a table constraint. In that case, there may be multiple local columns that refer to multiple columns in the referenced table. The referenced columns must be a multicolumn primary key, or they must otherwise have a multicolumn `UNIQUE` index. A foreign key definition can include several other optional parts. For the full syntax, see `CREATE TABLE` in Appendix C.

Unlike a table's own primary key, foreign keys are not required to be unique. This is because multiple foreign key values (multiple rows) in one table may refer to the same row in another table. This is known as a one-to-many relationship. Please see the section "One-to-Many Relationships" on page 95. Foreign keys are often marked with the abbreviation "FK," as shown in Figure 6-2.

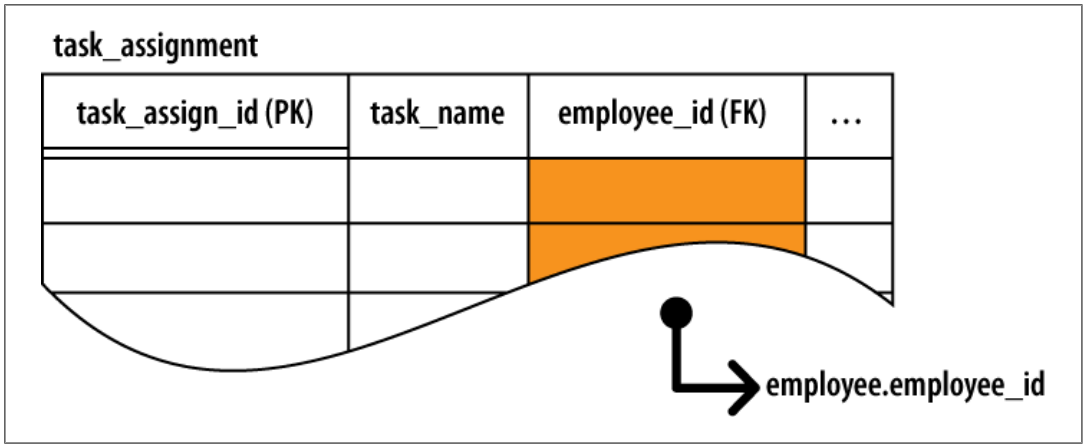

Figure 6-2. Foreign keys are copies of the primary key from another row. Foreign keys act as references or pointers to other rows. They are often identified with the abbreviation FK.

Foreign Key Constraints

Declaring foreign keys in the table definition allows the database to enforce foreign key constraints. *Foreign key constraints* are used to keep foreign key references in sync. Among other things, foreign key constraints can prevent "dangling references" by requiring that all foreign key values correctly match a row value from the columns of the referenced table. Foreign keys can also be set to NULL. A NULL clearly marks the foreign key as unassigned, which is a bit different than having an invalid value. In many cases, unassigned foreign keys don't fit the design of the database. In that case, the foreign key columns should be declared with a `NOT NULL` constraint.

Using our previous example, foreign key constraints would require that every `task_assignment.employee_id` element needs to contain the value of a valid `employee.employee_id`. By default, a NULL foreign key would also be allowed, but we've defined the `task_assignment.employee_id` column with a `NOT NULL` constraint. This demands that every task reference a valid employee.

Native foreign key support was added in SQLite 3.6.19, but is turned off by default. You must use the `PRAGMA foreign_keys` command to turn on foreign key constraints. This was done to avoid problems with existing applications and database files. A future version of SQLite may have foreign key constraints enabled by default. If your application is dependent on this setting, it should explicitly turn it on or off.

Modifications to either the foreign key table or the referenced table can potentially cause violations of the foreign key constraint. For example, if a statement attempted to update a `task_assignment.employee_id` value to an invalid `employee_id`, the foreign key constraint would be violated. Similarly, if an `employee` row was assigned a new `employee_id` value, any existing `task_assignment` references that point to the old value would become invalid. This would also violate the foreign key constraint.

If the database detects a change that would cause a foreign key violation, there are several actions that can be taken. The default action is to prohibit the change. For example, if you attempted to delete an employee that still had tasks assigned to them, the delete would fail. You would need to delete the tasks or transfer them to a different employee before you could delete the original employee.

Other conflict resolutions are also available. For example, using an `ON DELETE CASCADE` foreign key definition, deleting an employee would cause the database to automatically delete any tasks assigned to that employee. For more information on conflict resolutions and other advanced foreign key options, please see the SQLite website. Up-to-date documentation on SQLite's support for foreign keys can be found at *http://www.sqlite.org/foreignkeys.html*.

Correctly defining foreign keys is one of the most critical aspects of data integrity and security. Once defined, foreign key constraints make sure that data relationships remain valid and consistent.

Generic ID Keys

If you look at most database designs, you'll notice that a large number of the tables have a generic ID column that is used as the primary key. The ID is typically an arbitrary integer value that is automatically assigned as new rows are inserted. The number may be incrementally assigned, or it might be assigned from a sequence mechanism that is guaranteed to never assign the same ID number twice.

When an SQLite column is defined as an `INTEGER PRIMARY KEY`, that column will replace the hidden `ROWID` column that acts as the root column of every table. Using an `INTEGER PRIMARY KEY` allows for some significant performance enhancements. It also allows SQLite to automatically assign sequenced ID values. For more details, see "Primary keys" on page 40.

At first, a generic ID field might seem like a design cheat. If each table should have a specific and well-defined role, then the primary key should reflect what makes any given row unique—reflecting, in part, the essential definition of the items in a table. Using a generic and arbitrary ID to define that uniqueness seems to be missing the point.

From a theoretical standpoint, that may be correct, but this is one of those areas where theory bumps into reality, and reality usually wins.

Practically speaking, many datasets don't have a truly unique representation. For example, the names of people are not sufficiently unique to be used as database keys. Names are reasonably unique, and they do a fair job at identifying individuals in person, but they lack the inherent and complete uniqueness that good database design demands. Names also change from time to time, such as when people get married.

The more you dig around, the more you'll find that the world is full of data like this. Data that is sufficiently unique and stable for casual use, but not truly, absolutely

unique or fixed enough to use as a smart database key. In these situations, the best solution is to simply use an arbitrary ID that the database designer has total control over, even if it is meaningless outside of the database. This type of key is known as a *surrogate key*.

There are also situations when the primary key consists of three or four (or more!) columns. This is somewhat rare, but there are situations when it does come up. If you've got a large multicolumn primary key in the middle of a complex set of relationships, it can be a big pain to create and match all those multicolumn foreign keys. To simplify such situations, it is often easier to simply create an arbitrary ID and use that as the primary key.

Using an arbitrary ID is also useful if the customary primary key is physically large. Because each foreign key is a full copy of the primary key, it is unwise to use a lengthy text value or BLOB as the primary key. Rather, it would be better to use an arbitrary identifier and simply reference to the identifier.

One final comment on key names. There is often a temptation to name a generic ID field something simple, such as `id`. After all, if you've got an `employee` table, it might seem somewhat redundant to name the primary key `employee_id`; you end up with a lot of column references that read `employee.employee_id`, when it seems that `employee.id` is clear enough.

Well, by itself, it is clear enough, but primary keys tend to show up in other tables as foreign keys. While `employee.employee_id` might be slightly redundant, the name `task_assignment.employee_id` is not. That name also gives you significant clues about the column's function (a foreign key) and what table and column it references (the `employee_id` column, which is the PK column of the `employee` table). Using the same name for primary keys and foreign keys makes the inherent meaning and linkage a lot more obvious. It also allows shortcuts, such as the `NATURAL JOIN` or `JOIN...USING( )` syntax. Both of these forms require that matching columns have the exact same name.

Using a more explicit name also avoids the problem of having multiple tables, each with a column named `id`. Such a common name can make things somewhat confusing if you join together three or four tables. While I wouldn't necessarily prefix every column name, keeping primary key names unique within a database (and using the exact same name for foreign keys) can make the intent of the database design a lot more clear.

Keep It Specific

The biggest stumbling block for beginning database developers is that they don't create enough tables. Less experienced developers tend to view tables as large and monumental structures. There tends to be a feeling that tables are important, and that each table needs a fair number of columns containing significant amounts of data to justify its existence. If a design change calls for a new column or set of data points, there tends

to be a strong desire to lump everything together into large centralized tables rather than breaking things apart into logical groups.

The "tables must be big" mentality will quickly lead to poor designs. While you will have a few large, significant tables at the core of most designs, a typical design will also have a fair number of smaller auxiliary tables that may only hold two or three columns of data. In fact, in some cases you may find yourself building tables that consist of nothing but external references to other tables. Creating a new table is the only means you have to define a new data structure or data container, so any time you find yourself needing a new container, that's going to indicate a new table.

Every table should have a well-defined and clear role, but that doesn't always mean it will be big or stuffed with data.

Common Structures and Relationships

Database design has a small number of design structures and relationships that act as basic building blocks. Once you master how to use them and how to manipulate them, you can use these building blocks to build much larger and more complex data representations.

One-to-One Relationships

The most basic kind of inter-table relationship is the *one-to-one relationship*. As you can guess, this type of relationship establishes a reference from a single row in one table to a single row in another table. Most commonly, one-to-one relationships are represented by having a foreign key in one table reference a primary key in another table. If the foreign key column is made unique, only one reference will be allowed. As Figure 6-3 illustrates, a unique foreign key creates a one-to-one relationship between the rows of the two tables.

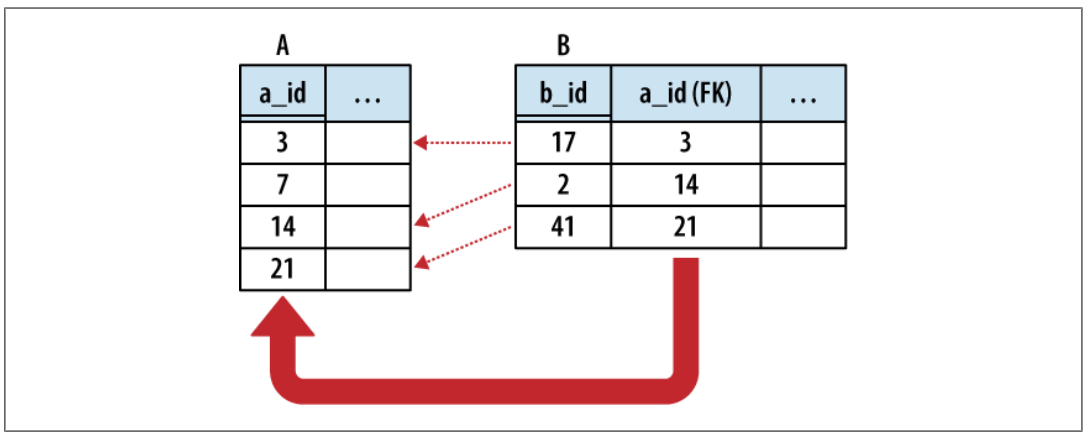

Figure 6-3. In a one-to-one relationship, table B has a foreign key that references the primary key of table A. This associates every non-NULL foreign key in table B with some row in table A.

In the strictest sense, a foreign key relationship is a one-to-(zero or one) relationship. When two tables are involved in a one-to-one relationship, there is nothing to enforce that every primary key has an incoming reference from a foreign key. For that matter, if the foreign key allows NULLs, there may be unassigned foreign keys as well.

One-to-one relationships are commonly used to create *detail tables*. As the name implies, a detail table typically holds details that are linked to the records in a more prominent table. Detail tables can be used to hold data that is only relevant to a small subsection of the database application. Breaking the detail data apart from the main tables allows different sections of the database design to evolve and grow much more independently.

Detail tables can also be helpful when extended data only applies to a limited number of records. For example, a website might have a `sales_items` table that lists common information (price, inventory, weight, etc.) for all available items. Type-specific data can then be held in detail tables, such as `cd_info` for CDs (artist name, album name, etc.) or `dvd_info` (directors, studio, etc.) for DVDs. Although the `sales_items` table would have a unique one-to-one relationship with every type-specific info table, each individual row in the `sales_item` table would be referenced by only one type of detail table.

One-to-one relationships can also be used to isolate very large data elements, such as BLOBs. Consider an employee database that contains an image of each employee. Due to the data storage and I/O overhead, it might be unwise to include a `photo` column directly in the `employee` table, but it is easy to create a photo table that references the employee table. Consider these two tables:

```
CREATE TABLE employee (
    employee_id  INTEGER   NOT NULL   PRIMARY KEY,
    name         TEXT   NOT NULL
    /* ...etc... */
);

CREATE TABLE employee_photo (
    employee_id  INTEGER   NOT NULL   PRIMARY KEY
                 REFERENCES employee,
    photo_data   BLOB
    /* ...etc... */
);
```

This example is a bit unique, because the `employee_photo.employee_id` column is both the primary key for the `employee_photo` table, as well as a foreign key to the `employee` table. Since we want a one-to-one relationship, it makes sense to just pair up primary keys. Because this foreign key does not allow NULL keys, every `employee_photo` row must be matched to a specific `employee` row. The database does not guarantee that every `employee` will have a matching `employee_photo`, however.

One-to-Many Relationships

One-to-many relationships establish a link between a single row in one table to multiple rows in another table. This is done by expanding a one-to-one relationship, allowing one side to contain duplicate keys. One-to-many relationships are often used to associate lists or arrays of items back to a single row. For example, multiple shipping addresses can be linked to a single account. Unless otherwise specified, "many" usually means "zero or more."

The only difference between a one-to-one relationship and a one-to-many relationship is that the one-to-many relationship allows for duplicate foreign key values. This allows multiple rows in one table (the many table) to refer back to a single row in another table (the One table). In building a one-to-many relationship, the foreign key must always be on the many side.

Figure 6-4 illustrates the relationship in more detail.

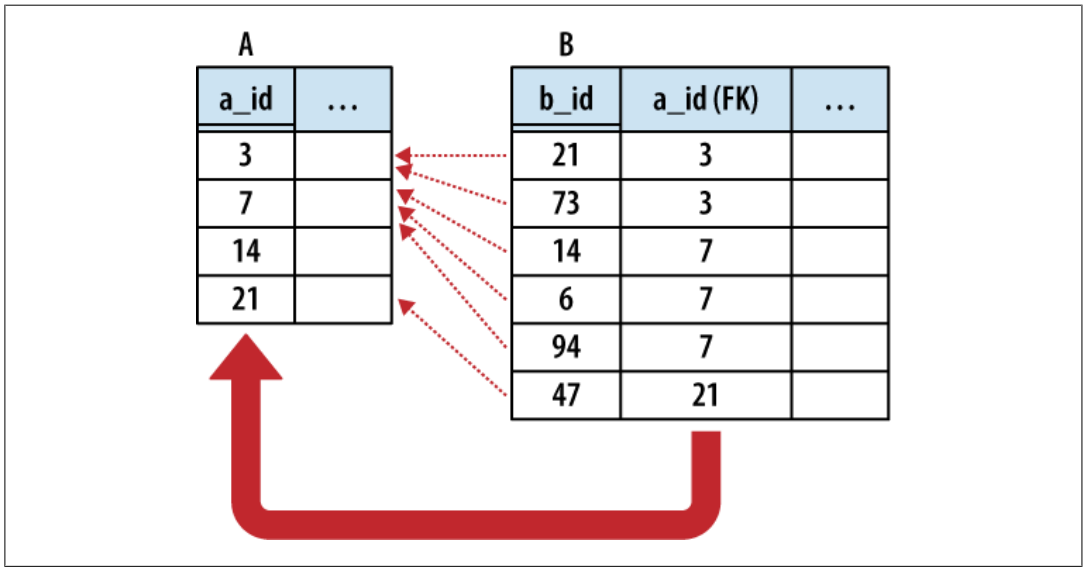

Figure 6-4. When building a one-to-many relationship, the primary keys must be unique, but foreign key columns may contain duplicate values. This means a one-to-many relationship must have the foreign key on the many side. Here we see a foreign key value in Table B refer to the primary key in Table A.

Any time you find yourself wondering how to stuff an array or list into the column of a table, the solution is to separate the array elements out into their own table and establish a one-to-many relationship.

If you need to represent a list or array, try using a one-to-many relationship.

The same is true any time you start to contemplate sets of columns, such as `item0`, `item1`, `item2`, etc., that are all designed to hold instances of the same type of value. Such designs have inherent limits, and the insert, update, and removal process becomes quite complex. It is much easier to just break the data out into its own table and establish a proper relationship.

One example of a one-to-many relationship is music albums, and the songs they contain. Each album has a list of songs associated with that album. For example:

```
CREATE TABLE albums (
        album_id INTEGER   NOT NULL   PRIMARY KEY,
        album_name TEXT );

CREATE TABLE tracks (
        track_id INTEGER   NOT NULL   PRIMARY KEY,
        track_name TEXT,
        track_number INTEGER,
        track_length INTEGER, -- in seconds
        album_id INTEGER   NOT NULL   REFERENCES albums );
```

Each album and track has a unique ID. Each track also has a foreign key reference back to its album. Consider:

```
INSERT INTO albums VALUES ( 1, "The Indigo Album" );
INSERT INTO tracks VALUES ( 1, "Metal Onion", 1, 137, 1 );
INSERT INTO tracks VALUES ( 2, "Smooth Snake", 2, 212, 1 );
INSERT INTO tracks VALUES ( 3, "Turn A", 3, 255, 1 );

INSERT INTO albums VALUES ( 2, "Morning Jazz" );
INSERT INTO tracks VALUES ( 4, "In the Bed", 1, 214, 2 );
INSERT INTO tracks VALUES ( 5, "Water All Around", 2, 194, 2 );
INSERT INTO tracks VALUES ( 6, "Time Soars", 3, 265, 2 );
INSERT INTO tracks VALUES ( 7, "Liquid Awareness", 4, 175, 2 );
```

To get a simple list of tracks and their associated album, we just join the tables back together. We can also sort by both album name and track number:

```
sqlite> SELECT album_name, track_name, track_number
   ...>   FROM albums JOIN tracks USING ( album_id )
   ...>   ORDER BY album_name, track_number;

album_name    track_name  track_number
------------  ----------  ------------
Morning Jazz  In the Bed  1
Morning Jazz  Water All   2
Morning Jazz  Time Soars  3
Morning Jazz  Liquid Awa  4
The Indigo A  Metal Onio  1
The Indigo A  Smooth Sna  2
The Indigo A  Turn A      3
```

We can also manipulate the track groupings:

```
sqlite> SELECT album_name, sum( track_length ) AS runtime, count(*) AS tracks
   ...>   FROM albums JOIN tracks USING ( album_id )
   ...>   GROUP BY album_id;
```

```
album_name        runtime     tracks
----------------  ----------  ----------
The Indigo Album  604         3
Morning Jazz      848         4
```

This query groups the tracks based off their album, and then aggregates the track data together.

Many-to-Many Relationships

The next step is the many-to-many relationship. A *many-to-many relationship* associates one row in the first table to many rows in the second table while simultaneously allowing individual rows in the second table to be linked to multiple rows in the first table. In a sense, a many-to-many relationship is really two one-to-many relationships built across each other.

Figure 6-5 shows the classic many-to-many example of people and groups. One person can belong to many different groups, while each group is made up of many different people. Common operations are to find all the groups a person belongs to, or to find all the people in a group.

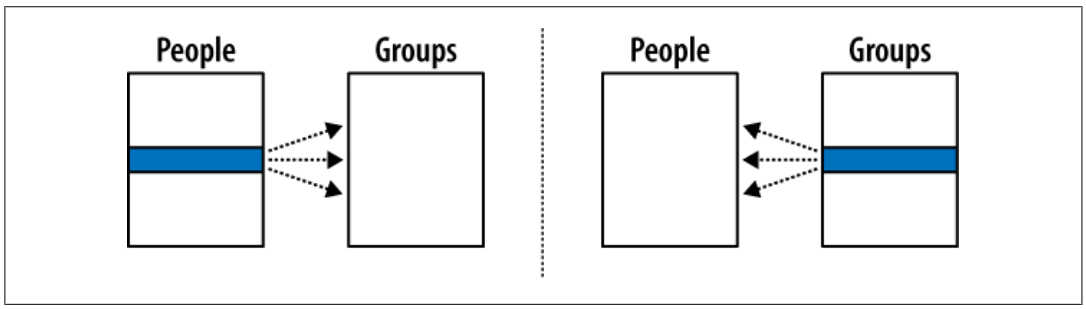

Figure 6-5. A many-to-many relationship is like two one-to-many relationships built across each other. In this example, each individual person can be a member of one or more groups, while each group can contain one or more people.

Many-to-many relationships are a bit more complex than other relationships. Although the tables have a many-to-many relationship with each other, the entries in both tables must remain unique. We cannot duplicate either person rows or group rows for the purpose of matching keys. This is a problem, since each foreign key can only reference one row. This makes it impossible for a foreign key of one table (such as a group) to refer to multiple rows of another table (such as people).

To solve this, we go back to the advice from before: if you need to add a list to a row, break out that list into its own table and establish a one-to-many relationship with the new table. You cannot directly represent a many-to-many relationship with only two tables, but you can take a pair of one-to-many relationships and link them together. The link requires a small table, known as a *link table*, or *bridge table*, that sits between the two many tables. Each many-to-many relationship requires a unique bridge table.

In its most basic form, the bridge table consists of nothing but two foreign keys—one for each of the tables it is linking together. Each row of the bridge table links one row in the first many table to one row in the second many table. In our People-to-Groups example, the link table defines a membership of one person in one group. This is illustrated in Figure 6-6.

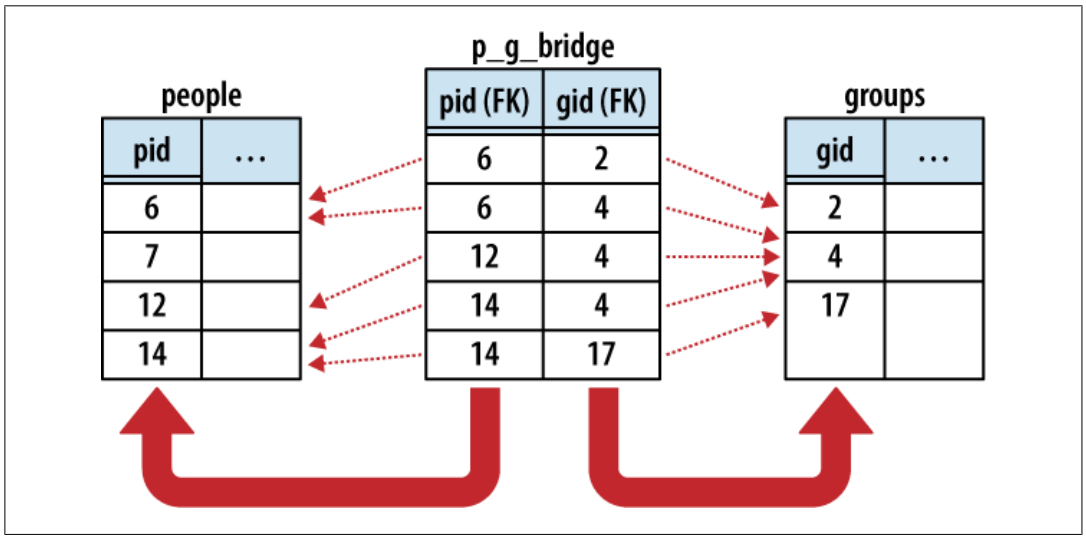

Figure 6-6. Implementing a many-to-many relationship requires a bridge table. In this example, each row of the bridge table represents a membership of one person in one group. Note that the primary key of the bridge table is a multicolumn key over (p_id, g_id). This keeps memberships unique.

The logical representation of a many-to-many relationship is actually a one-to-many-to-one relationship, with the bridge table (acting as a record of membership) in the middle. Each person has a one-to-many relationship with memberships, just as groups have a one-to-many relationship with memberships. In addition to binding the two tables together, the bridge table can also hold additional information about the membership itself, such as a first-joined date or an expiration date.

Here are three tables. The `people` and `groups` table are obvious enough. The `p_g_bridge` table acts as the bridge table between the `people` table and `groups` table. The two columns are both foreign key references, one to `people` and one to `groups`. Establishing a primary key across both foreign key columns ensures the memberships remain unique:

```
CREATE TABLE people ( pid INTEGER   PRIMARY KEY, name TEXT, ... );
CREATE TABLE groups ( gid INTEGER   PRIMARY KEY, name TEXT, ... );
CREATE TABLE p_g_bridge(
        pid INTEGER   NOT NULL   REFERENCES people,
        gid INTEGER   NOT NULL   REFERENCES groups,
        PRIMARY KEY ( pid, gid )
);
```

This query will list all the groups that a person belongs to:

```
SELECT groups.name AS group_name
  FROM people JOIN p_g_bridge USING ( pid ) JOIN groups USING ( gid )
  WHERE people.name = search_person_name;
```

The query simply links `people` to `groups` using the bridge table, and then filters out the appropriate rows.

We don't always need all three tables. This query counts all the members of a group without utilizing the `people` table:

```
SELECT name AS group_name, count(*) AS members
  FROM groups JOIN p_g_bridge USING ( gid )
  GROUP BY gid;
```

There are many other queries we can do, like finding all the groups that have no members:

```
SELECT name AS group_name
  FROM groups LEFT OUTER JOIN p_g_bridge USING ( gid )
  WHERE pid IS NULL;
```

This query performs an outer join from the `groups` table to the `p_g_bridge` table. Any unmatched group rows will be padded out with a NULL in the `p_g_bridge.pid` column. Since this column is marked `NOT NULL`, we know the only possible way for a row to be NULL in that column is from the outer join, meaning the row is unmatched to any memberships. A very similar query could be used to find any people that have no memberships.

Hierarchies and Trees

Hierarchies and other tree-style data relationships are common and frequently show up in database design. Modeling one in a database can be a challenge because you tend to ask different types of questions when dealing with hierarchies.

Common tree operations include finding all the sub-nodes under a given node, or querying the depth of a given node. These operations have an inherent recursion in them—a concept that SQL doesn't support. This can lead to clumsy queries or complex representations.

There are two common methods for representing a tree relation using database tables. The first is the *adjacency model*, which uses a simple representation that is easy to modify but complex to query. The other common representation is the *nested set*, which allows relatively simple queries, but at the cost of a more complex representation that can be expensive to modify.

Adjacency Model

A tree is basically a series of nodes that have a one-to-many relationship between the parents and the children. We already know how to define a one-to-many relationship: give each child a foreign key that points to the primary key of the parent. The only trick is that the same table is sitting on both sides of the relationship.

For example, here is a basic adjacency model table:

```
CREATE TABLE tree (
    node   INTEGER   NOT NULL   PRIMARY KEY,
    name   TEXT,
    parent INTEGER   REFERENCES tree );
```

Each node in the tree will have a unique node identifier. Each node will also have a reference to its parent node. The root of the tree can simply have a NULL parent reference. This allows multiple trees to be stored in the same table, as multiple nodes can be defined as a root node of different trees.

If we want to represent this tree:

```
A
  A.1
    A.1.a
  A.2
    A.2.a
    A.2.b
  A.3
```

We would use the following data:

```
INSERT INTO tree VALUES ( 1, 'A',     NULL );
INSERT INTO tree VALUES ( 2, 'A.1',   1 );
INSERT INTO tree VALUES ( 3, 'A.1.a', 2 );
INSERT INTO tree VALUES ( 4, 'A.2',   1 );
INSERT INTO tree VALUES ( 5, 'A.2.a', 4 );
INSERT INTO tree VALUES ( 6, 'A.2.b', 4 );
INSERT INTO tree VALUES ( 7, 'A.3',   1 );
```

The following query will give a list of nodes and parents by joining the tree table to itself:

```
sqlite> SELECT n.name AS node, p.name AS parent
   ...>   FROM tree AS n JOIN tree AS p ON n.parent = p.node;

node        parent
----------  ----------
A.1         A
A.1.a       A.1
A.2         A
A.2.a       A.2
A.2.b       A.2
A.3         A
```

Inserting or removing nodes is fairly straightforward, as is moving subtrees around to different parents.

What isn't easy are tree-centric operations, like counting all of the nodes in a subtree, or computing the depth of a specific node (often used for output formatting). You can do limited traversals (like finding a grand-parent) by joining the tree table to itself multiple times, but you cannot write a single query that can compute an answer to these types of questions on a tree of arbitrary size. The only choice is to write application routines that loop over different levels in the tree, computing the answers you seek.

Overall, the adjacency model is easy to understand, and the trees are easy to modify. The model is based on foreign keys, and can take full advantage of the database's built-in referential integrity. The major disadvantage is that many types of common data queries require the application code to loop over several individual database queries and assist in calculating answers.

Nested set

As the name implies, the nested set representation depends on nesting groups of nodes inside other groups. Rather than representing some type of parent-child relationship, each node holds bounding data about the full subtree underneath it. With some clever math, this allows us to query all kinds of information about a node.

A nested set table might look like this:

```
CREATE TABLE nest (
    name  TEXT,
    lower INTEGER   NOT NULL   UNIQUE,
    upper INTEGER   NOT NULL   UNIQUE,
    CHECK ( lower < upper ) );
```

The nested set can be visualized by converting the tree structure into a parenthetical representation. We then count the parentheses and record the upper and lower bound of each nested set. The index numbers can also be calculated with a depth-first tree traversal, where the lower bound is a pre-visit count and the upper bound is a post-visit count.

```
A( A.1( A.1.a( ) ), A.2( A.2.a( ), A.2.b( )  ), A.3(  )  )
 1    2      3 4 5     6      7 8       9 10 11    12 13 14
```

Or, in SQL:

```
INSERT INTO nest VALUES ( 'A',      1, 14 );
INSERT INTO nest VALUES ( 'A.1',    2,  5 );
INSERT INTO nest VALUES ( 'A.1.a',  3,  4 );
INSERT INTO nest VALUES ( 'A.2',    6, 11 );
INSERT INTO nest VALUES ( 'A.2.a',  7,  8 );
INSERT INTO nest VALUES ( 'A.2.b',  9, 10 );
INSERT INTO nest VALUES ( 'A.3',   12, 13 );
```

This might not look that useful, but it allows a number of different queries. For example, if you want to find all the leaf nodes (nodes without children) just look for nodes that have an upper and lower value that are next to each other:

```
SELECT name FROM nest WHERE lower + 1 = upper;
```

You can find the depth of a node by counting all of its ancestors:

```
sqlite> SELECT n.name AS name, count(*) AS depth
   ...>   FROM nest AS n JOIN nest AS p
   ...>     ON p.lower <= n.lower AND p.upper >= n.upper
   ...>   GROUP BY n.name;

name        depth
----------  ----------
A           1
A.1         2
A.1.a       3
A.2         2
A.2.a       3
A.2.b       3
A.3         2
```

There are many other queries that match patterns or differences in the upper and lower bounds.

Nested sets are very efficient at calculating many types of queries, but they are expensive to change. For the math to work correctly, there can't be any gaps in the numbering sequence. This means that any insert or delete requires renumbering a significant number of entries in the table. This isn't bad for something with a few dozen nodes, but it can quickly prove impractical for a tree with hundreds of nodes.

Additionally, because nested sets aren't based off any kind of key reference, the database can't help enforce the correctness of the tree structure. This leaves database integrity and correctness in the hands of the application—something that is normally avoided.

More information

This is just a brief overview of how to represent tree relationships. If you need to implement a tree, I suggest you do a few web searches on *adjacency model* or *nested set*. Many of the larger SQL books mentioned in "Wrap-up" on page 58 also have sections on tree and hierarchies.

Normal Form

You won't get too far into most database design books without reading a discussion on normalization and the Normal Forms. The *Normal Forms* are a series of forms, or table design specifications, that describe the best way to lay out data in a database. The higher the normal form, the more normalized the database is. Each form builds on the previous one, adding additional rules or conditions that must be met. *Normalization* is the process of removing data duplication, more clearly defining key relationships, and generally moving towards a more idealized database form. It is possible for different tables in the same database to be at different levels.

Most people recognize five normal forms simply referred to as the First Normal Form through the Fifth Normal Form. These are often abbreviated 1NF through 5NF. There are also a few named forms, such as the Boyce-Codd Normal Form (BCNF). Most of these other forms are roughly equivalent to one of the numbered forms. For example, BCNF is a slight extension to the Third Normal Form. Some folks also recognize higher levels of normalization, such as a Sixth Normal Form and beyond, but these extreme levels of normalization are well beyond the practical concerns of most database designers.

Normalization

The normalization process is useful for two reasons. First, normalization specifies design criteria that can act as a guide in the design process. If you have a set of tables that are proving to be difficult to work with, that often points to a deeper design problem or assumption. The normalization process provides a set of rules and conditions that can help identify trouble spots, as well as provide possible solutions to reorganize the data in a more consistent and clean fashion.

The other advantage, which shows up more at runtime, is that data integrity is much easier to enforce and maintain in a normalized database. Although the overall database design is often more complex (i.e., more tables), the individual parts are usually much simpler and fill more clearly defined roles. This often translates to better `INSERT`, `UPDATE`, and `DELETE` performance, since changes are often smaller and more localized.

Localizing data is a core goal of data normalization. Most of the normal forms deal with eliminating redundant or replicated data so that each unique token of data is stored once—and only once—in the database. Everything else simply references that definitive copy. This makes updates easier, since there is only one place an update needs to be applied, but it also makes the data more consistent, as it is impossible for multiple copies of the data to become out of sync with each other. When working on a schema design, a question you should constantly ask yourself is, "If this piece of data changes, how many different places will I need to make that change?" If the answer is anything other than one, chances are you're not in Normal Form.

Denormalization

Normalizing a database and spreading the data out into different tables means that queries usually involve joining several tables back together. This can occasionally lead to performance concerns, especially for complex reports that require data from a large number of tables. These concerns can sometimes lead to the process of *denormalization*, where duplicate copies of the same data are intentionally introduced to reduce the number of joins required for common queries. This is typically done on systems that are primarily read-only, such as data-warehouse databases, and is often done by computing temporary tables from properly normalized source data.

While a large number of joins can lead to performance concerns, database optimization is just like code optimization—don't start too early and don't make assumptions. In general, the advantages of normalization far outweigh the costs. A correct database that runs a tad slower is infinitely more useful than a very fast database that returns incorrect or inconsistent answers.

The First Normal Form

The *First Normal Form*, or *1NF*, is the lowest level of normalization. It is primarily concerned with making sure a table is in the proper format. There are three conditions that must be met for a table to be in 1NF.

The first condition relates to ordering. To be in 1NF, the individual rows of a table cannot have any meaningful or inherent order. Each row should be an isolated, stand-alone record. The meaning of a value in one row cannot depend on any of the data values from neighboring rows, either by insertion order, or by some sorted order. This condition is usually easy to meet, as SQL does not guarantee any kind of row ordering.

The second condition is uniqueness. Every row within a 1NF table must be unique, and must be unique by those columns that hold meaningful data for the application. For example, if the only difference between two rows is the database-maintained `ROWID` column, then the rows aren't really unique. However, it is perfectly fine to consider an arbitrary sequence ID (such as an `INTEGER PRIMARY KEY`) to be part of the application data. This condition establishes that the table must have some type of `PRIMARY KEY`, consisting of one or more columns that creates a unique definition of what the table represents.

The third and final condition for 1NF requires that every column of every row holds one (and only one) logical value that cannot be broken down any further. The concern is not with compound types, such as dates (which might be broken down into integer day, month, and year values) but with arrays or lists of logical values. For example, you shouldn't be recording a text value that contains a comma-separated list of logical, independent values. Arrays or lists should be broken out into their own one-to-many relationships.

The Second Normal Form

The *Second Normal Form*, or *2NF*, deals with compound keys (multicolumn keys) and how other columns relate to such keys. 2NF has only one condition: every column that is not part of the primary key must be relevant to the primary key as a whole, and not just a sub-part of the key.

Consider a table that lists all the conference rooms at a large corporate campus. At minimum, the `conf_room` table has columns for `building_numb` and `room_numb`. Taken together, these two columns will uniquely identify any conference room across the whole campus, so that will be our compound primary key.

Next, consider a column like `seating_capacity`. The values in this column are directly dependent on each specific conference room. That, by definition, makes the column dependent on both the building number and the room number. Including the `seating_capacity` column will not break 2NF.

Now consider a column like `building_address`. This column is dependent on the `building_numb` column, but it is not dependent on the `room_numb` column. Since `building_address` is dependent on only part of the primary key, including this column in the `conf_room` table would break 2NF.

Because 2NF is specifically concerned with multicolumn keys, any table with a single-column primary key that is in 1NF is automatically in 2NF.

To recognize a column that might be breaking 2NF, look for columns that have duplicate values. If the duplicate values tend to line up with duplicate values in one of the primary key columns, that is a strong indication of a problem. For example, the `building_address` column will have a number of duplicate values (assuming most buildings have more than one conference room). The duplicate address values can be matched to duplicate values in the `building_numb` column. This alignment shows how the address column is tied to only the `building_numb` column specifically, and not the whole primary key.

The Third Normal Form

The *Third Normal Form*, or *3NF*, extends the 2NF to eliminate transitive key dependencies. A transitive dependency is when A depends on B, and B depends on C, and therefore A depends on C. 3NF requires that each nonprimary key column has a direct (nontransitive) dependency on the primary key.

For example, consider an inventory database that is used to track laptops at a small business. The `laptop` table will have a primary key that uniquely identifies each laptop, such as an inventory control number. It is likely the table would have other columns that include the make and model of the machine, the serial number, and perhaps a purchase date. For our example, the `laptop` table will also include a `responsible_person_id` column. When an employee is assigned a laptop, their employee ID number is put in this column.

Within a row, the value of the `responsible_person_id` column is directly dependent on the primary key. In other words, each individual laptop is assigned a specific responsible person, making the values in the `responsible_person_id` column directly dependent on the primary key of the `laptop` table.

Now consider what happens when we add a column like `responsible_person_email`. This is a column that holds the email address of the responsible person. The value of this column is still dependent on the primary key of the `laptop` table. Each individual laptop has a specific `responsible_person_email` field that is just as unique as the `responsible_person_id` field.

The problem is that the values in the `responsible_person_email` column are not *directly* dependent on an individual laptop. Rather, the email column is tied to the `responsible_person_id`, and the `responsible_person_id` is, in turn, dependent on the individual laptop. This transitive dependency breaks 3NF, indicating that the `respon sible_person_email` column doesn't belong there.

In the `employee` table, we will also find both a `person_id` column and an `email` column. This is perfectly acceptable if the `person_id` is the primary key (likely). That would make the `email` column directly dependent on the primary key, keeping the table in 3NF.

A good way to recognize columns that may break 3NF is to look for pairs or sets of unrelated columns that need to be kept in sync with each other. Consider the `laptop` table. If a system was reassigned to a new person, you would always update both the `responsible_person_id` column and the `responsible_person_email` column. The need to keep columns in sync with each other is a strong indication of a dependency to each other, rather than to the primary key.

Higher Normal Forms

We're not going to get into the details of BCNF, or the Fourth or Fifth (or beyond) Normal Forms, other than to mention that the Fourth and Fifth Normal Forms start to deal with inter-table relationships and how different tables interact with each other. Most database designers make a solid effort to get everything into 3NF and then stop worrying about it. It turns out that if you get the hang of things and tend to turn out table designs that are in 3NF, chances are pretty good that your tables will also meet the conditions for 4NF and 5NF, if not higher. To a large extent, the higher Normal Forms are formal ways of addressing some edge cases that are somewhat unusual, especially in simpler designs.

Although the conditions of the Normal Forms build on each other, the typical design process doesn't actually iterate over the individual Forms. You don't sit down with a new design and alter it until everything is 1NF, just to turn around and muck with the design until everything is 2NF, and so on, in a isolated step-by-step manner. Once you understand the ideas and concepts behind the First, Second, and Third Normal Forms, it becomes second nature to design directly to 3NF. Stepping over the conditions one at a time can help you weed out especially difficult trouble spots, but it doesn't take long to gain a sense of when a design looks clean and when something "just ain't right."

The core concept to remember is that each table should try to represent one and only one thing. The primary key(s) for that table should uniquely and inherently identify the concept behind the table. All other columns should provide supporting data specific to that one concept. When speaking of the first three Normal Forms in a 1982 CACM article, William Kent wrote that each non-key column " . . . must provide a fact about the key, the whole key, and nothing but the key." If you incorporate only one formal aspect of database theory into your designs, that would be a great place to start.

Indexes

Indexes (or indices) are auxiliary data structures used by the database system to enforce unique constraints, speed up sort operations, and provide faster access to specific records. They are often created in hopes of making queries run faster by avoiding table scans and providing a more direct lookup method.

Understanding where and when to create indexes can be an important factor in achieving good performance, especially as datasets grow. Without proper indexes in place, the database system has no option but to do full table scans for every lookup. Table scans can be especially expensive when joining tables together.

Indexes are not without cost, however. Each index must maintain a one-to-one correspondence between index entries and table rows. If a new row is inserted, updated, or deleted from a table, that same modification must be made to all associated indexes. Each new index will add additional overhead to the `INSERT`, `UPDATE`, and `DELETE` commands. Indexes also consume space, both on disk as well as in the SQLite page cache. A proper, well-placed index is worth the cost, but it isn't wise to just create random indexes, in hopes that one of them will prove useful. A poorly placed index still has all the costs, and can actually slow queries down. You can have too much of a good thing.

How They Work

Each index is associated with a specific table. Indexes can be either single column or multicolumn, but all the columns of a given index must belong to the same table. There is no limit to the number of indexes a table can have, nor is there a limit on the number of indexes a column can belong to. You cannot create an index on a view or on a virtual table.

Internally, the rows of a normal table are stored in an indexed structure. SQLite uses a B-Tree for this purpose, which is a specific type of multi-child, balanced tree. The details are unimportant, other than understanding that as rows are inserted into the tree, the rows are sorted, organized, and optimized, so that a row with a specific, known `ROWID` can be retrieved relatively directly and quickly.

When you create an index, the database system creates another tree structure to hold the index data. Rather than using the `ROWID` column as the sort key, the tree is sorted and organized using the column or columns you've specified in the index definition. The index entry consists of a copy of the values from each of the indexed columns, as well as a copy of the corresponding `ROWID` value. This allows an indexed entry to be found very quickly using the values from the indexed columns.

If we have a table like this:

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

And then create an index that looks like this:

```
CREATE INDEX idx_tbl_a_b ON tbl ( a, b );
```

The database generates an internal data structure that is conceptually similar to:

```
SELECT a, b, ROWID FROM tbl ORDER BY a, b;
```

If SQLite needs to quickly find all the rows where, for example, `a = 45`, it can use the sorted index to quickly jump to that range of values and extract the relevant index entries. If it's looking for the value of `b`, it can simply extract that from the index and be done. If we need any other value in the row, it needs to fetch the full row. This is done by looking up the `ROWID`. The last value of any index entry is the `ROWID` of its corresponding table row. Once SQLite has a list of `ROWID` values for all rows where `a = 45`, it can efficiently look up those rows in the original table and retrieve the full row. If everything works correctly, the process of looking up a small set of index entries, and then using those to look up a small set of table rows, will be much faster and more efficient than doing a full table scan.

Must Be Diverse

To have a positive impact on query performance, indexes must be diverse. The cost of fetching a single row through an index is significantly higher than fetching a single row through a table scan. To reduce the overall cost, an index must overcome this overhead by eliminating the vast majority of row fetches. By significantly reducing the number of row lookups, the total query cost can be reduced, even if the individual row lookups are more expensive.

The break-even point for index performance is somewhere in the 10% to 20% range. Any query that fetches more rows from a table will do better with a table scan, while any query that fetches fewer rows will see an improvement by using an index. If an index cannot isolate rows to this level, there is no reason for it to be there. In fact, if the query optimizer mistakenly uses an index that returns a large percentage of rows, the index can reduce performance and slow things down.

This creates two concerns. First, if you're trying to improve the performance of a query that fetches a moderate percentage of rows, adding an index is unlikely to help. An index can only improve performance if it is used to target a focused number of rows.

Second, even if the query is asking for a specific value, this can still lead to a higher percentage of fetched rows if the indexed columns are not reasonably unique. For example, if a column only has four unique values, a successful query will never fetch fewer than approximately 25% of the rows (assuming a reasonable distribution of values or queries). Adding an index to this type of column will not improve performance. Creating an index on a true/false column would be even worse.

When the query optimizer is trying to figure out if it should use an index or not, it typically has very little information to go on. For the most part, it assumes that if an

index exists, it must be a good index, and it will tend to use it. The `ANALYZE` command can help mitigate this by providing statistical information to the query optimizer (see `ANALYZE` for more details), but keeping that updated and well maintained can be difficult in many environments.

To avoid problems, you should avoid creating indexes on columns that are not reasonably unique. Indexing such columns rarely provides any benefit, and it can confuse the query optimizer.

INTEGER PRIMARY KEYs

When you declare one or more columns to be a `PRIMARY KEY`, the database system automatically creates a unique index over those columns. The fundamental purpose of this index is to enforce the `UNIQUE` constraint that is implied with every `PRIMARY KEY`. It also happens that many database operations typically involve the primary key, such as natural joins, or conditional lookups in `UPDATE` or `DELETE` commands. Even if the index wasn't required to enforce the `UNIQUE` constraint, chances are good you would want an index over those columns anyway.

There are further advantages to using an `INTEGER PRIMARY KEY`. As we've discussed, when an `INTEGER PRIMARY KEY` is declared, that column replaces the automatic `ROWID` column, and becomes the root column for the tree. In essence, the column becomes the index used to store the table itself, eliminating the need for a second, external index.

`INTEGER PRIMARY KEY` columns also provide better performance than standard indexes. `INTEGER PRIMARY KEY`s have direct access to the full set of row data. A normal index simply references the `ROWID` value, which is then looked up in the table root. `INTEGER PRIMARY KEY`s can provide indexed lookups, but skip this second lookup step, and directly access the table data.

If you're using generic row ID values, it is worth the effort to define them as `INTEGER PRIMARY KEY` columns. Not only will this reduce the size of the database, it can make your queries faster and more efficient.

Order Matters

When creating a multicolumn index for performance purposes, the column order is very important. If an index has only one column, that column is used as the sort key. If additional columns are specified in the index definition, the additional columns will be used as secondary, tertiary, etc., sort keys. All of the data is sorted by the first column, then any groups of duplicate values are further sorted by the second column, and so on. This is very similar to a typical phonebook. Most phone books sort by last name. Any group of common last names is then sorted by first name, and then by middle name or initial.

In order to utilize a multicolumn index, a query must contain conditions that are able to utilize the sort keys in the same order they appear in the index definition. If the query does not contain a condition that keys off the first column, the index cannot be used.

Consider looking up a name in the phonebook. You can use a phonebook to quickly find the phone number for "Jennifer T. Smith." First, you look up the last name "Smith." Then you refine the search, looking for the first name, "Jennifer," and finally the middle initial "T" (if required). This sequence should allow you to focus in on a specific entry very quickly.

A phonebook isn't much use to quickly find all the people with the first name "Jennifer." Even though the first name is part of the phonebook index, it isn't the first column of the index. If our query cannot provide a specific condition for the first column of a multicolumn index, the index cannot be used, and our only option is to do a full scan. At that point it is usually more efficient to do a full scan on the table itself, rather than the index.

It is not necessary to utilize every column, only that you utilize the columns in order. If you're looking up a very unique name in the phone book, you may not need to search off the first name or middle initial. Or, perhaps your query is to find every "Smith." In that case, the conditions of the query are satisfied with just the first column.

In the end, you need to be very careful when considering the order of a multicolumn index. The additional columns should be viewed as a refinement on the first column, not a replacement. A single multicolumn index is very different from multiple single-column indexes. If you have different queries that use different columns as the main lookup condition, you may be better off with multiple small indexes, rather than one large multicolumn index.

One at a Time

Multicolumn indexes are very useful in some situations, but there are limitations on when they can be effectively used. In some cases, it is more appropriate to create multiple single-column indexes on the same table.

This too has limitations. You cannot create a single-column index on every column of a table and expect the query system to mix and match whatever indexes it needs. Once a subset of rows is extracted from a table (via index or full table scan), that set of rows conceptually becomes an independent temporary table that is no longer part of the original table. At that point, any index associated with the source table becomes unavailable.

In general, the query optimizer can only utilize one index per table-instance per query. Multiple indexes on a single table only make sense if you have a different queries that extract rows using different columns (or have multiple `UNIQUE` constraints). Different queries that key off different columns can pick the index that is most appropriate, but

each individual query will only use one index per table-instance. If you want a single query to utilize multiple indexed columns, you need a multicolumn index.

The major exception is a series of `OR` conditions. If a `WHERE` clause has a chain of `OR` conditions on one or more columns of the same table, then there are cases when SQLite may go ahead and use multiple indexes for the same table in a single query.

Index Summary

For all their power, indexes are often a source of great frustration. The right index will result in a huge performance boost, while the wrong index can significantly slow things down. All indexes add cost to table modifications and add bulk to the database size. A well-placed index is often worth the overhead costs, but it can be difficult to understand what makes a well-placed index.

To complicate matters, you don't get to tell the query optimizer when or how to use your indexes—the query planner makes its own decisions. Having everything be automatic means that queries will work no matter what, but it can also make it difficult to determine if an index is being used. It can be even more difficult to figure out why an index is being ignored.

This essentially leaves the database designer in the position of second-guessing the query optimizer. Changes and optimizations must be searched for on something of a trial-and-error basis. To make things worse, as your application changes and evolves, changes in the query structure or patterns can cause a change in index use. All in all, it can be a difficult situation to manage.

Traditionally, this is where the role of the *DBA*, or *Database Administrator*, comes in. This person is responsible for the care and feeding of a large database server. They do things like monitor query efficiency, adjust tuning parameters and indexes, and make sure all the statistical data is kept up to date. Unfortunately, that whole idea is somewhat contrary to what SQLite is all about.

So how do you maintain performance? For starters, have a solid design. A well-crafted and normalized database is going to go a long way toward defining your access patterns. Taking the time to correctly identify and define your primary keys will allow the database system to create appropriate indexes and give the query optimizer (and future developers) some insight into your design and intentions.

Also remember that (with the exception of `UNIQUE` constraints) a database will operate correctly without any indexes. It might be slow, but all the answers will be correct. This lets you worry about design first, and performance later. Once you have a working design, it is always possible to add indexes after the fact, without needing to alter your table structure or your query commands.

The queries of a fully normalized database will typically have quite a few `JOIN` operations. Nearly all of these joins are across primary and foreign keys. Primary keys are

automatically indexed, but in most cases you need to manually create indexes on your foreign keys. With those in place, your database should already have the most critical indexes defined.

Start from here. Measure the performance of the different types of queries your application uses and find the problem areas. Looking at these queries, try to find columns where an index might boost performance. Look for any constraints and conditions that may benefit from an index, especially in join conditions that reference nonkey columns. Use `EXPLAIN` and `EXPLAIN QUERY PLAN` to understand how SQLite is accessing the data. These commands can also be used to verify if a query is using an index or not. For more information, see `EXPLAIN` in Appendix C. You can also use the `sqlite3_stmt_status()` function to get a more measured understanding of statement efficiency. See `sqlite3_stmt_status()` in Appendix G for more details.

All in all, you shouldn't get too hung up on precise index placement. You may need a few well-placed indexes to deal with larger tables, but the stock `PRIMARY KEY` indexes do surprisingly well in most cases. Further index placement should be considered performance tuning and shouldn't be worried about until an actual need is identified. Never forget that indexes have cost, so you shouldn't create them unless you can identify a need.

Transferring Design Experience

If you have some experience designing application data structures or class hierarchies, you may have noticed some similarities between designing runtime structures and database tables. Many of the organization principles are the same and, thankfully, much of the design knowledge and experience gained in the application development world will transfer to the database world.

There are differences, however. Although these differences are minor, they often prove to be significant stumbling blocks for experienced developers that are new to database design. With a little insight, we can hopefully avoid the more common misconceptions, opening up the world of database design to those with existing experience in data structure and class design.

Tables Are Types

The most common misconception is to think of tables as *instances* of a compound data structure. A table looks a whole lot like an array or a dynamic list, so it is easy to make this mistake.

Tables should be thought of as type definitions. You should never use a named table as a data organizer or record grouping. Rather, each table definition should be treated like a data structure definition or a class definition. SQL DDL commands such as `CREATE TABLE` are conceptually similar to those C/C++ header files that define an application's

data structures and classes. The table itself should be considered a global management pool for all instances of that type. If you need a new instance of that type, you simply insert a new row. If you need to group or catalog sets of instances, do that with key associations, not by creating new tables.

If you ever find yourself creating a series of tables with identical definitions, that's usually a big warning. Any time your application uses string manipulation to programmatically build table names, that's also a big warning—especially if the table names are derived from values stored elsewhere in the database.

Keys Are Backwards Pointers

Another stumbling block is the proper use of keys. Keys are very similar to pointers. A primary key is used to identify a unique instance of a data structure. This is similar to the address of a record. Anything that wants to reference that record needs to record its address as a pointer or, in the cases of databases, as a foreign key. Foreign keys are essentially database pointers.

The trick is that database references are backwards. Rather than pointers that indicate ownership ("I manage that"), foreign keys indicate a type of possession ("I am managed by that").

In C or C++, if a main data record manages a list of sub-records, the main data record would have some kind of pointer list. Each pointer would reference a specific sub-record associated with this main record. If you are dealing with the main data record and need the list of sub-records, you simply look at the pointer list.

Databases do it the other way around. In a one-to-many relationship, the main record (the "one" side row) would simply have a primary key. All the sub-records (the "many" side rows) would have foreign keys that point back at the main record. If you are dealing with the main record and need the list of sub-records, you ask the database system to look at the global pool of all subrecord instances and return just those subrecords that are managed by this main record.

This tends to make application developers uncomfortable. This is not the way traditional programming language data structures tend to be organized, and the idea of searching a large global record pool just to retrieve a small number of records tends to raise all kinds of performance concerns. Thankfully, this is exactly the kind of thing that databases are very good at doing.

Do One Thing

My final design advice is more general. As with data structures or classes, the fundamental idea behind a table is that it should represent instances of one single idea or "thing." It might represent a set of nouns or things, such as people, or it might represent verbs or actions, such as a transaction log. Tables can even represent less concrete

things, such as a many-to-many bridge table that records membership. But no matter what it is, each table should have one, and *only* one, clearly defined role in the database. Normally the problem isn't too many tables, it is too few.

If the meaning of one field is ever dependent on the value of another field, the design is heading in a bad direction. Two different meanings should have two different tables (or two different columns).

Closing

As with application development, database design is part science and part art. It may seem quite complex, with keys to set up, different relationships to define, Normal Forms to follow, and indexes to create.

Thankfully, the basics usually fall into place fairly quickly. If you start to get into larger or more complex designs, some reading on more formal methods of data modeling and database design might be in order, but most developers can get pretty far by just leveraging their knowledge and experience in designing application data structures. In the end, you're just defining data structures and hooking them together.

Finally, don't expect to get it right the first time. Very often, when going through the database design process, you realize that your understanding of the thing you're trying to store is incorrect or incomplete. Just as you might refactor a class hierarchy, don't be afraid to refactor a database design. It should be an evolving thing, just like your application code and data structures.