Learn SQL The Hard Way

Contents

Preface

This is a simple course that teaches you the basics of SQL. You'll learn the core commands and syntax for creating, reading, updating, and deleting data. Along with the book there is a small set of videos that demonstrate the exercises and so you can see how these commands work in real life. I find this combination of text with videos helps people retain the information better than with simply text or videos alone.

The best way to work through this book is to do the following:

1. Read an exercise and take notes about the contents.
2. Attempt to do the exercise yourself, solving any problems you possibly can on your own.
3. Watch the video for additional clues and information.
4. Write down any questions you have or problems you faced before moving on to the next exercise.

This is fundamentally the way all of my books work. They're designed so that you learn how to solve problems on your own using reasoning and debugging skills. By learning the core concepts through rote practice and then applying them to creative problems you will learn quicker.

This book also takes things slow. You aren't given a huge wall of text full of jargon and then expected to type a few lines of code to understand the concepts. Instead you are given a small amount of code, a discussion of the code, and then a demonstration of the concepts when you're ready to understand them. If you're the kind of person who need to understand large conceptual structures full of mathematical proofs before you feel you can write single line of code then my books are not for you.

Finally, this book won't make you a grand master SQL database administrator or data modeling expert. There's a vast trove of books that already teach these concepts. This book is just enough knowledge to get started with those other books. At the end of this book I'll give you a list of these other books so you can continue your studies of SQL and data modeling in general.

If you run into problems please feel free to email me at help@learncodethehardway.org any time.

Introduction: Haters Gonna Hate, Or Why You Still Need SQL

SQL is everywhere, and I'm not saying that because I want you to use it. It's just a fact. I bet you have some in your pocket right now. All Android Phones and iPhones have easy access to a SQL database called SQLite and many applications on your phone use it directly. It runs banks, hospitals, universities, governments, small businesses, large ones, just about every computer and every person on the planet eventually touches something running SQL. SQL is an incredibly successful and solid technology.

The problem with SQL is it seems *everyone* hates its guts. It is a weird obtuse kind of "non-language" that most programmers can't stand. It was designed long before any of these modern problems like "web scale" or Object Oriented Programming even existed. Despite being based on a solid mathematically built theory of operation, it gets enough wrong to be annoying. Trees? Nested objects and parent child relationships? SQL just laughs in your face and gives you a massive flat table saying, "You figure it out loser."

Why should you learn SQL if everyone hates it so much? Because behind this supposed hate is a lack of understanding of what SQL is and how to use it. The NoSQL movement is partially a reaction to antiquated database servers, and also a response to a fear of SQL borne from ignorance of how it works. By learning SQL, you actually will learn important theoretical concepts that apply to nearly every data storage system past and present.

No matter what the SQL haters claim, you should learn SQL because it is everywhere, and it's actually not that hard to learn enough to be educated about it. Becoming an educated SQL user will help you make informed decisions about what databases to use, whether to not use SQL, and give you a deeper understanding of many of the systems you work with as a programmer.

Ultimately though, I want you to learn SQL because it is very handy. I can use SQLite to prototype a simple data model an application and I am confident it will work just about everywhere that has SQLite. This ability to use a cross-platform consistent and powerful data storage language is very valuable.

About This Book

This book teaches SQL to anyone, but it helps if you can code already. The concepts in SQL are taught assuming you at least know how to do some programming even if it's just a tiny bit. It also assumes you can run commands from the command line, know how to use the shell, and have access to a good programmer's text editor.

To keep the book simple, and since managing a giant database server is tangential to learning SQL, this book will use SQLite3 to teach the fundamentals of the language. It will use SQLite3 similar to how you

use Python or Ruby and you'll be writing full `.sql` scripts and running them to learn the language.

When you're done with this book, you should understand the basics of SQL, how to get a simple database server running, a bit about data design, and you should be able to branch out into another database of your choice.

What Is SQL?

I pronounce SQL "sequel" but you can also say "ESS-QUEUE-ELL" if you want. SQL also stands for Structured Query Language but by now nobody even cares about that since that was just a marketing ploy anyway. What SQL does is give you a language for interacting with data in a database. It's advantage though is that it closely matches a theory established many years ago defining properties of well structured data. It's not exactly the same (which some detractors lament) but it's close enough to be useful.

How SQL works is it understands fields that are in tables, and how to find the data in the tables based on the contents of the fields. All SQL operations are then one of four general things you do to tables:

Create Putting data into tables (or create tables).

Read Query data out of a table.

Update Change data already in a table.

Delete Remove data from the table.

This has been given the acronym "CRUD" and is considered a fundamental set of features *every* data storage system must have. In fact, if you can't do one of these four in some way then there better be a very good reason.

One way I like to explain how SQL works is by comparing it to a spreadsheet software like Excel:

- A database is a whole spreadsheet file.
- A table is a tab/sheet in the spreadsheet, with each one being given a name.
- A column is a column in both.
- A row is a row in both.
- SQL then gives you a language for doing CRUD operations on these to produce *new tables or alter existing ones*.

The last item is significant, and not understanding this causes people a lot of headaches. SQL only knows tables, and every operation produces tables. It either "produces" a table by modifying an existing one, or it returns a new temporary table as your data set.

As you read this book, you'll begin to understand the significance of this design. For example, one of the reasons Object Oriented languages are mismatched with SQL databases is that OOP languages are organized around graphs, but SQL wants only returns tables. Since it's possible to map nearly any graph to a table this works, but it places a lot of work on the OOP language to do the translation.

Another place that causes a mismatch is in SQL concepts such as ternary relationships and attributed relationships, which OOP completely does not understand. In SQL I can make 3 tables related to each other using a 4th table, and that 4th table is a cohesive relationship. To do the same thing in an OOP language I have to make a whole intermediary class that encodes this relationship, which is kind of weird in OOP.

This may sound like total magic incantations right now, but by the time you're done with this book you'll understand these issues and how to deal with them.

Against Indoctrination

You may run into someone who thinks you should learn technology X because it is superior. They'll claim that learning SQLite will cripple you for life because it is missing features. Or, they may say SQL is dead and NoSQL is the future.

The problem is these people are trying to indoctrinate you, not educate you. They think of the world of technology as a zero sum game that they have to win, and if you learn SQL or SQLite then you won't learn their system of choice. Typically, this indoctrination is designed so that you must depend on their company's software and services to stay alive.

I want to educate you so that you have the ability to make your own choices and learn anything you want. I'm only using SQLite because it's the simplest and most complete SQL system you can install. Other servers are a huge pain to install and manage, but SQLite is one download, it's free, and it's credible. That's the *only* reason I'm using SQLite. I always advocate using the right tool for the job, and SQLite is the right tool for this job.

My only goal is to educate you on this particular tool in the simplest way I can so that you can improve as a programmer, and do it in such a way that you do *not* need to depend on me or my services when you're done.

License And Status

This book is the first edition and is very new. You may run into errors, and something may not flow well, but I can fix them very quickly if you find them. Any time you have a problem, just email me at help@learncodethehardway.org and I will fix it.

The Setup

This book will use SQLite3 as a training tool. SQLite3 is a complete database system that has the advantage of requiring almost no setup. You just download a binary and work it like most other scripting languages. Using this, you'll be able to learn SQL without getting stuck in the administrivia of administering a database server.

Installing SQLite3 is easy:

- Either go to their downloads page and grab the binary for your platform. Look for "Precompiled Binaries for X" with X being your operating system of choice.
- Use your operating system's package manager to install it. If you're on Linux, then you know what that means. If you're on OSX then first go get a package manager and then use it to install sqlite3.

When you've installed it, then make sure you can start up a command line and run it. Here's a quick test for you to try:

```
$ sqlite3 test.db
SQLite version 3.7.8 2011-09-19 14:49:19
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> create table test (id);
sqlite> .quit
```

Then look to see that the `test.db` file is there. If that works, then you're all set. You should make sure that your version of SQLite3 is the same as the one I have here: 3.7.8. Sometimes things won't work right with older versions.

Additional Tools You'll Need

You will also need to have the following additional tools:

- A good plain text editor. Use anyone you like, but do *not* use an IDE (Integrated Development Environment). They cannot help you.
- Familiarity with your command line (aka Terminal, aka cmd.exe). You'll be running commands from there.
- An internet connection with a web browser so you can look up documentation and research things I tell you to find.

Once you have that all setup, you are ready to go.

Video Installation Instructions

There is one video for each platform included with this book that you can watch to learn how to install what you need. There really is not much to install, since you just need SQLite3, a little Python, and a text editor. The videos are more for people who are just starting out and not as familiar with their computers. To use the Ex0 videos do this:

1. Find the video for your platform and watch that first. It will be named `LSQL_Debian_Ex0.mp4`, `LSQL_OSX_Ex0.mp4`, or `LSQL_Windows_Ex0.mp4`.

2. Watch that video and follow along to learn how to install what you'll need.

3. After you have everything installed for your platform, watch `LSQL_Ex0_Final.mp4` for additional setup instructions that work on all platforms.

Debian Exercise 0

There is a special video that shows you how to install a *complete* Debian Linux virtual machine with all the tools from scratch. If you are interested in eventually becoming a Database Administrator then I highly recommend you go through this video and the Command Line Crash Course (see below). Doing this will be difficult if you have no Linux system administration experience, but most database systems are running on Linux these days, so learning how to set it up and work with it is very useful. Even Microsoft's cloud platform supports Linux, Windows now supports Linux, and they have fully embraced Open Source.

Command Line Crash Course

You can read the Command Line Crash Course if you need to learn how to use the command line. You don't need to be a Unix master, but basic knowledge is necessary for most database administration these days. The Command Line is also a good tiny step into programming as it gets you into "talking" to your computer using commands.

Study Drills

- Go to the SQLite3 site again and browse around through the documentation.

EXERCISE 1

Creating Tables

In the introduction I said that you can do "Create Read Update Delete" operations to the data inside tables. How do you make the tables in the first place? By doing CRUD on the database *schema*, and the first SQL statement to learn is `CREATE`:

ex1.sql

```
1  CREATE TABLE person (
2      id INTEGER PRIMARY KEY,
3      first_name TEXT,
4      last_name TEXT,
5      age INTEGER
6  );
```

You could put this all on one line, but I want to talk about each line so it's on multiple ones. Here's what each line does:

ex1.sql:1 The start of the "CREATE TABLE" which gives the name of the table as `person`. You then put the fields you want inside parenthesis after this setup.

ex1.sql:2 An `id` column which will be used to exactly identify each row. The format of a column is `NAME TYPE`, and in this case I'm saying I want an `INTEGER` that is also a `PRIMARY KEY`. Doing this tells SQLite3 to treat this column special.

ex1.sql:3-4 A `first_name` and a `last_name` column which are both of type `TEXT`.

ex1.sql:5 An `age` column that is just a plain `INTEGER`.

ex1.sql:6 Ending of the list of columns with a closing parenthesis and then a semi-colon ';' character.

What You Should See

The easiest way to run this is to simply do: `sqlite3 ex1.db < ex1.sql` and it should just exit and not show you anything. To make sure it created a database use `ls -l`:

Exercise 1 Session

```
1  $ ls -l
2  total 16
3  -rw-r--r--  1 zedshaw  staff  2048 Nov  8 16:18 ex1.db
```

```
4 -rw-r--r--  1 zedshaw  staff    92 Nov  8 16:14 ex1.sql
```

Study Drills

- SQL is *mostly* a case-insensitive language. It was created in an era when case sensitivity was perceived as a major usability problem, so it has this quirk which can anoy the hell out of programmers from other languages. Rewrite this so that it's all lowercase and see if it still works. You'll need to delete ex1.db.
- Add other `INTEGER` and `TEXT` fields for other things a person might have.

Portability Notes

The types used by SQLite3 are usually the same as other databases, but be careful as one of the ways SQL database vendors differentiated themselves was to "embrace and extend" certain data types. The worst of these is anything to do with date and time.

EXERCISE 2

Creating A Multi-Table Database

Creating one table isn't too useful. I want you to now make 3 tables that you can store data into:

ex2.sql

```
CREATE TABLE person (
    id INTEGER PRIMARY KEY,
    first_name TEXT,
    last_name TEXT,
    age INTEGER
);

CREATE TABLE pet (
    id INTEGER PRIMARY KEY,
    name TEXT,
    breed TEXT,
    age INTEGER,
    dead INTEGER
);

CREATE TABLE person_pet (
    person_id INTEGER,
    pet_id INTEGER
);
```

In this file you are making tables for two types of data, and then "linking" them together with a third table. People call these "linking" tables "relations", but very pedantic people with no lives call all tables "relations" and enjoy confusing people who just want to get their jobs done. In my book, tables that have data are "tables", and tables that link tables together are called "relations".

There isn't anything new here, except when you look at `person_pet` you'll see that I've made two columns: `person_id` and `pet_id`. How you would link two tables together is simply *insert* a row into `person_pet` that had the values of the two row's id columns you wanted to connect. For example, if `person` contained a row with `id=20` and `pet` had a row with `id=98`, then to say that person owned that pet, you would insert `person_id=20, pet_id=98` into the `person_pet` relation (table).

We'll get into actually inserting data like this in the next few exercises.

What You Should See

You run this SQL script in the same way as before, but you specify `ex2.db` instead of `ex1.db`. As usual there's no output, but this time I want you to open the database and use the `.schema` command to dump it:

Exercise 2 Session

```
sqlite> .schema
CREATE TABLE person (
    id INTEGER PRIMARY KEY,
    first_name TEXT,
    last_name TEXT,
    age INTEGER
);
CREATE TABLE person_pet (
    person_id INTEGER,
    pet_id INTEGER
);
CREATE TABLE pet (
    id INTEGER PRIMARY KEY,
    name TEXT,
    breed TEXT,
    age INTEGER,
    dead INTEGER
);
sqlite>
```

The "schema" should match what you typed in.

Study Drills

- In these tables I made a 3rd relation table to link them. How would you get rid of this relation table `person_pet` and put that information right into `person`? What's the implication of this change?
- If you can put one row into `person_pet`, can you put more than one? How would you record a crazy cat lady with 50 cats? * Create another table for the cars people might own, and create its corresponding relation table.
- Search for "sqlite3 datatypes" in your favorite search engine and go read the "Datatypes In SQLite Version 3" document. Take notes on what types you can use and other things that seem important. We'll cover more later.

Portability Notes

Databases have a lot of options for specifying the keys in these relations, but for now we'll keep it simple.

Inserting Data

You have a couple tables to work with, so now I'll have you put some data into them using the `INSERT` command:

ex3.sql

```
1  INSERT INTO person (id, first_name, last_name, age)
2      VALUES (0, 'Zed', 'Shaw', 37);
3
4  INSERT INTO pet (id, name, breed, age, dead)
5      VALUES (0, 'Fluffy', 'Unicorn', 1000, 0);
6
7  INSERT INTO pet VALUES (1, 'Gigantor', 'Robot', 1, 1);
```

In this file I'm using two different forms of the `INSERT` command. The first form is the more explicit style, and most likely the one you should use. It specifies the columns that will be inserted, followed by `VALUES`, then the data to include. Both of these lists (column names and values) go inside parenthesis and are separated by commas.

The second version on line 7 is an abbreviated version that doesn't specify the columns and instead relies on the implicit order in the table. This form is dangerous since you don't know what column your statement is actually accessing, and some databases don't have reliable ordering for the columns. It's best to only use this form when you're really lazy.

What You Should See

I'm going to reuse the `ex2.sql` file from the previous exercise to recreate the database so you can put data into it. This is what it looks like when I run it:

Exercise 3 Session

```
1  $ sqlite3 ex3.db < ex2.sql
2  $ sqlite3 -echo ex3.db < ex3.sql
3  INSERT INTO person (id, first_name, last_name, age)
4      VALUES (0, "Zed", "Shaw", 37);
5  INSERT INTO pet (id, name, breed, age, dead)
6      VALUES (0, "Fluffy", "Unicorn", 1000, 0);
```

```
7 INSERT INTO pet VALUES (1, "Gigantor", "Robot", 1, 1);
8 $
```

In the first line I just make `ex3.db` from the `ex2.sql` file. Then I add the `-echo` argument to `sqlite3` so that it prints out what it is doing. After that the data is in the database and ready to query.

Study Drills

- Insert yourself and your pets (or imaginary pets like I have).
- If you changed the database in the last exercise to not have the `person_pet` table then make a new database with that schema, and insert the same information into it.
- Go back to the list of data types and take notes on what format you need for the different types. For example, how many ways can you write TEXT data.

Portability Notes

As I mentioned in the last exercise, database vendors tend to add lock-in to their platforms by extending or altering the data types used. They'll subtly make *their* TEXT columns a little different here, or *their* DATETIME columns are called TIMESTAMP and take a different format. Watch out for this when you use a different database.

EXERCISE 4

Insert Referential Data

In the last exercise you filled in some tables with people and pets. The only thing that's missing is who owns what pets, and that data goes into the `person_pet` table like this:

ex4.sql

```
INSERT INTO person_pet (person_id, pet_id) VALUES (0, 0);
INSERT INTO person_pet VALUES (0, 1);
```

Again I'm using the explicit format first, then the implicit format. How this works is I'm using the `id` values from the person row I want (in this case, 0) and the `id` from the pet rows I want (again, 0 for the Unicorn and 1 for the Dead Robot). I then insert one row into `person_pet` relation table for each "connection" between a person and a pet.

When you create a table like this you are creating a "relation". Technically every table is a "relation" but I find that confusing because only some tables are actually used to relate (connect) tables to other tables. The `person_pet` table allows you to relate the data in `person` to the data in `pet`. Later you'll learn to link these tables with special queries called "joins" that connect one table, to another, using equality. In this book I will only cover the "baby" version of joins, relations, and foreign keys because that topic is more advanced and also becomes less standardized between databases.

What You Should See

I'll just piggyback on the last exercise and run this right on the `ex3.db` database to set these values:

Exercise 4 Session

```
$ sqlite3 -echo ex3.db < ex4.sql
INSERT INTO person_pet (person_id, pet_id) VALUES (0, 0);
INSERT INTO person_pet VALUES (0, 1);
$
```

Study Drills

- Add the relationships for you and your pets.

- Using this table, could a pet be owned by more than one person? Is that logically possible? What about the family dog? Wouldn't everyone in the family technically own it?
- Given the above, and given that you have an alternative design that puts the `pet_id` in the `person` table, which design is better for this situation?

Selecting Data

Out of the CRUD matrix you only know "Create". You can create tables and you can create rows in those tables. I'll now show you how to "Read" or in the case of SQL, `SELECT`:

ex5.sql

```
1  SELECT * FROM person;
2
3  SELECT name, age FROM pet;
4
5  SELECT name, age FROM pet WHERE dead = 0;
6
7  SELECT * FROM person WHERE first_name != 'Zed';
```

Here's what each of these lines does:

ex5.sql:1 This says "select all columns from person and return all rows." The format for `SELECT` is `SELECT what FROM tables(s) WHERE (tests)` and the `WHERE` clause is optional. The '`*`' (asterisk) character is what says you want all columns.

ex5.sql:3 In this one I'm only asking for two columns `name` and `age` from the `pet` table. It will return all rows.

ex5.sql:5 Now I'm looking for the same columns from the `pet` table but I'm asking for *only* the rows where `dead = 0`. This gives me all the pets that are alive.

ex5.sql:7 Finally I'm selecting all columns from `person` just like in the first line, but now I'm saying only if they do *not* equal "Zed". That `WHERE` clause is what determines which rows to return or not.

What You Should See

When you run this with `sqlite3 -echo` you should get something like the following output:

Exercise 5 Session

```
1  $ sqlite3 -echo ex3.db < ex5.sql
2  SELECT * FROM person;
3  0|Zed|Shaw|37
4  SELECT name, age FROM pet;
```

```
Fluffy|1000
Gigantor|1
SELECT name, age FROM pet WHERE dead = 0;
Fluffy|1000
SELECT * FROM person WHERE first_name != "Zed";
$
```

I say "something like" because if you were doing the extra credit this whole time you will have different rows in your database. For example, if you added yourself then you will have some rows listed at the end. In my example above I have nothing returned for the last query because I'm the only person in the `person` table, and that means no row match the last query's `WHERE` clause. Study this carefully.

Study Drills

- Write a query that finds all pets older than 10 years.
- Write a query to find all people younger than you. Do one that's older.
- Write a query that uses more than one test in the `WHERE` clause using the AND to write it. For example, `WHERE first_name = "Zed" AND age > 30`.
- Do another query that searches for rows using 3 columns and uses both `AND` and `OR` operators.

Portability Notes

Some databases have additional operators and boolean logic tests, but just stick to the regular ones that you find in most programming languages for now.

EXERCISE 6

Join Many Tables

Hopefully you're getting your head around selecting data out of tables. Always remember this: *SQL ONLY KNOWS TABLES. SQL LOVES TABLES. SQL ONLY RETURNS TABLES. TABLES. TABLES. TABLES. TABLES!* I repeat this in this rather crazy manner so that you will start to realize that what you know in programming isn't going to help. In programming, you deal in graphs and in SQL you deal in tables. They're related concepts, but the mental model is different.

Here's an example of where it becomes different. Imagine you want to know what pets Zed owns. You need to write a `SELECT` that looks in `person` and then "somehow" finds Zed's pets. To do that you have to query the `person_pet` table to get the `id` columns you need. To do that you need to either *join* the three tables (`person`, `person_pet`, `pet`) together with equality expressions, *or* you have to do a second select to return the correct `pet.id` numbers.

Here's how I'd do it with a join:

ex6.sql

```
1  /* normal join with equality */
2  SELECT pet.id, pet.name, pet.age, pet.dead
3      FROM pet, person_pet, person
4      WHERE
5      pet.id = person_pet.pet_id AND
6      person_pet.person_id = person.id AND
7      person.first_name = 'Zed';
8
9  /* using a sub-select */
10 SELECT pet.id, pet.name, pet.age, pet.dead
11     FROM pet
12     WHERE pet.id IN
13     (
14         SELECT pet_id FROM person_pet, person
15         WHERE person_pet.person_id = person.id
16         AND person.first_name = 'Zed'
17     );
```

Now this looks like a lot, but I'll break it down so you can see it's simply crafting a new table based on data in the three tables and the `WHERE` clause:

ex6.sql:1 A simple comment telling you what this query does.

ex6.sql:2 I only want some columns from `pet` so I specify them in the select. In the last exercise you used '*' to say "every column" but that's going to be a bad idea here. Instead, you want to be explicit

and say what column from each table you want, and you do that by using `table.column` as in `pet.name`.

ex6.sql:3 To connect `pet` to `person` I need to go through the `person_pet` relation table. In SQL that means I need to list all three tables after the `FROM`.

ex6.sql:4 Start the `WHERE` clause.

ex6.sql:5 First I connect `pet` to `person_pet` by the related id columns `pet.id` and `person_pet.id`.

ex6.sql:6 *AND* I need to connect `person` to `person_pet` in the same way. Now the database can search for only the rows where the id columns all match up, and those are the ones that are connected.

ex6.sql:7 *AND* I finally ask for only the pets that I own by adding a `person.first_name` test for my first name.

In the first SQL `SELECT` I'm joining the three tables by setting different `id` columns equal. This links them together so that the rows "line up" and are connected. The second `SELECT` then uses a sub-select to do the same thing:

ex6.sql:9 A comment again telling you what's going on.

ex6.sql:10 The exact same start to the `SELECT` as on line 2.

ex6.sql:11 However, we only want data from the `pet` table in the "master select" because we'll be using the `IN` keyword to start a sub-select to get the `pet.id` values we need.

ex6.sql:12 The `WHERE` clause then says `pet.id IN` which tells SQLite3 that we are going to take the `pet.id` values we need from another SQL query.

ex6.sql:13 I then start this sub-select for the `IN` with a parenthesis.

ex6.sql:14 I now need to *only* get the right `pet_id` values from `person` and `person_pet` using a simpler join. Just like with any `SELECT` I list out the columns I want and what tables they are `FROM`.

ex6.sql:15 I need a where clause that sets the equality needed, but I only need to worry about `person.id` being matched up with `person_pet.person_id`.

ex6.sql:16 `AND` finally my name `"Zed"` to get just the animals I own.

ex6.sql:17 Then we close off this sub-select with `)` and end the whole SQL statement with `;`.

What You Should See

I rebuild the database again using all the `.sql` files I've made so far and then run the queries two ways:

Exercise 6 Session

```
$ rm ex6.db
$ sqlite3 ex6.db < code/ex2.sql
$ sqlite3 ex6.db < code/ex3.sql
$ sqlite3 ex6.db < code/ex4.sql
$ sqlite3 ex6.db < code/ex6.sql
0|Fluffy|1000|0
1|Gigantor|1|1
0|Fluffy|1000|0
1|Gigantor|1|1
$ sqlite3 -column -header ex6.db < code/ex6.sql
id          name        age         dead
----------  ----------  ----------  ----------
0           Fluffy      1000        0
1           Gigantor    1           1
id          name        age         dead
----------  ----------  ----------  ----------
0           Fluffy      1000        0
1           Gigantor    1           1
```

If you don't get exactly the same data, then do a `SELECT` on the `person_pet` table and make sure it's right. You might have inserted too many values into it.

Study Drills

- This may be a mind blowing weird way to look at data if you already know a language like Python or Ruby. Take the time to model the same relationships using classes and objects then map it to this setup.
- Do a query that finds your pets you've added thus far.
- Change the queries to use your `person.id` instead of the `person.name` like I've been doing.

Portability Notes

There are actually other ways to get these kinds of queries to work called "joins". I'm avoiding those concepts for now because they are *insanely* confusing. Just stick to this way of joining tables for now and ignore people who try to tell that this is somehow slower or "low class".

Deleting Data

This is the simplest exercise, but I want you to think for a second before typing the code in. If you had "SELECT * FROM" for SELECT, and "INSERT INTO" for INSERT, then how would you write the DELETE format? You can probably glance down but try to guess at what it would be then look.

ex7.sql

```
1  /* make sure there's dead pets */
2  SELECT name, age FROM pet WHERE dead = 1;
3
4  /* aww poor robot */
5  DELETE FROM pet WHERE dead = 1;
6
7  /* make sure the robot is gone */
8  SELECT * FROM pet;
9
10 /* let's resurrect the robot */
11 INSERT INTO pet VALUES (1, 'Gigantor', 'Robot', 1, 0);
12
13 /* the robot LIVES! */
14 SELECT * FROM pet;
```

I'm simply implementing a very complex update of the robot by deleting him and then putting the record back but with dead=0. In later exercises I'll show you how to use UPDATE to do this, so don't consider this to be the real way you'd do an update.

Most of the lines in this script are already familiar to you, except for line 5. Here you have the DELETE and it has nearly the same format as other commands. You give DELETE FROM table WHERE tests and a way to think about it is being like a SELECT that removes rows. Anything that works in a WHERE clause will work here.

What You Should See

I'm going to reconstruct the entire database from scratch by replaying all the exercises to this point that you need. This shows you how your work so far should continue to work as you go through the exercises.

Exercise 7 Session

```
$ rm ex7.db
$ sqlite3 ex7.db < ex2.sql
$ sqlite3 ex7.db < ex3.sql
$ sqlite3 ex7.db < ex4.sql
$ sqlite3 -echo ex7.db < ex7.sql
SELECT name, age FROM pet WHERE dead = 1;
Gigantor|1
DELETE FROM pet WHERE dead = 1;
SELECT * FROM pet;
0|Fluffy|Unicorn|1000|0
INSERT INTO pet VALUES (1, "Gigantor", "Robot", 1, 0);
SELECT * FROM pet;
0|Fluffy|Unicorn|1000|0
1|Gigantor|Robot|1|0
$
```

Notice at the end I'm adding the `sqlite3 -echo` so you can see what statements run and what they produce.

Study Drills

- Go through the output from your run and make sure you know what table is produced for which SQL commands and how they produced that output.
- Combine all of `ex2.sql` through `ex7.sql` into one file and redo the above script so you just run this one new file to recreate the database.
- At the top of this new `.sql` file, add `DROP TABLE` commands to drop the tables you're about to recreate. Now your script can run without you needing to `rm ex3.db`. You'll need to go look up the syntax for `DROP TABLE`.
- Add onto the script to delete other pets and insert them again with new values. Remember that this is *not* how you normally update records and is only for the exercise.

Deleting Using Other Tables

Remember I said, "DELETE is like SELECT but it removes rows from the table." The limitation is you can only delete from one table at a time. That means to delete all the pets you need to do some additional queries and then delete based on those.

One way you do this is with a sub-query that selects the ids you want delete based on a query you've already written. There are other ways to do this, but this is one you can do right now based on what you know:

ex8.sql

```
1  DELETE FROM pet WHERE id IN (
2      SELECT pet.id
3      FROM pet, person_pet, person
4      WHERE
5      person.id = person_pet.person_id AND
6      pet.id = person_pet.pet_id AND
7      person.first_name = 'Zed'
8  );
9
10 SELECT * FROM pet;
11 SELECT * FROM person_pet;
12
13 DELETE FROM person_pet
14     WHERE pet_id NOT IN (
15         SELECT id FROM pet
16     );
17
18 SELECT * FROM person_pet;
```

The lines 1-8 are a DELETE command that starts off normally, but then the WHERE clause uses IN to match id columns in pet to the table that's returned in the sub-query. The sub-query (also called a sub-select) is then a normal SELECT and it should look really similar to the ones you've done before when trying to find pets owned by people.

On lines 13-16 I then use a sub-query to clear out the person_pet table of any pets that don't exist anymore by using NOT IN rather than IN.

What You Should See

I've changed the formatting on this and removed extra output that isn't relevant to this exercise. Notice how I'm using a new database called `mydata.db` and I'm using a conglomerate SQL file named `code.sql` that has all the SQL from exercises 2 through 7 in it. This makes it easier to rebuild and run this exercise. I'm also using `sqlite3 -header -column -echo` to get nicer output for the tables and to see the SQL that's being run. To create the `code.sql` file you can use any text editor you want, or do this:

```
cat ex2.sql ex3.sql ex4.sql ex7.sql > code.sql
```

Once you have the `code.sql` file this should work the same:

Exercise 8 Session

```
$ sqlite3 mydata.db < code.sql
# ... cut the output for this ...
$ sqlite3 -header -column -echo mydata.db < ex8.sql
DELETE FROM pet WHERE id IN (
    SELECT pet.id
    FROM pet, person_pet, person
    WHERE
    person.id = person_pet.person_id AND
    pet.id = person_pet.pet_id AND
    person.first_name = "Zed"
);

SELECT * FROM pet;

SELECT * FROM person_pet;
person_id   pet_id
----------  ----------
0           0
0           1

DELETE FROM person_pet
    WHERE pet_id NOT IN (
        SELECT id FROM pet
    );

SELECT * FROM person_pet;
$
```

You should see that after you `DELETE` the `SELECT` returns nothing.

Study Drills

- Practice writing `SELECT` commands and then put them in a `DELETE WHERE IN` to remove those records found. Try deleting any dead pets owned by you.
- Do the inverse and delete people who have dead pets.
- Do you really need to delete dead pets? Why not just remove their relationship in `person_pet` and mark them dead? Write a query that removes dead pets from `person_pet`.

Portability Notes

Depending on the database, sub-select will be slow.

Updating Data

You now know the CRD parts of CRUD, and I just need to teach you the Update part to round out the core of SQL. As with all the other SQL commands the `UPDATE` command follows a format similar to `DELETE` but it changes the columns in rows instead of deleting them.

ex9.sql

```
UPDATE person SET first_name = 'Hilarious Guy'
    WHERE first_name = 'Zed';

UPDATE pet SET name = 'Fancy Pants'
    WHERE id=0;

SELECT * FROM person;
SELECT * FROM pet;
```

In the above code I'm changing my name to "Hilarious Guy", since that's more accurate. And to demonstrate my new moniker I renamed my Unicorn to "Fancy Pants". He loves it.

This shouldn't be that hard to figure out, but just in case I'm going to break the first one down:

- Start with `UPDATE` and the table you're going to update, in this case `person`.
- Next use `SET` to say what columns should be set to what values. You can change as many columns as you want as long as you separate them with commas like `first_name = "Zed", last_name = "Shaw"`.
- Then specify a `WHERE` clause that gives a `SELECT` style set of tests to do on each row. When the `UPDATE` finds a match it does the update and `SET`s the columns to how you specified.

What You Should See

I'm resetting the database with my `code.sql` script and then running this:

Exercise 9 Session

```
$ sqlite3 mydata.db < code.sql
# ... output cut ...
$
```

```
$ sqlite3 -header -column -echo mydata.db < ex9.sql
UPDATE person SET first_name = "Hilarious Guy"
    WHERE first_name = "Zed";
UPDATE pet SET name = "Fancy Pants"
    WHERE id=0;

SELECT * FROM person;
id          first_name     last_name   age
----------  -------------  ----------  ----------
0           Hilarious Guy  Shaw        37

SELECT * FROM pet;
id          name         breed       age         dead
----------  -----------  ----------  ----------  ----------
0           Fancy Pants  Unicorn     1000        0
1           Gigantor     Robot       1           0
$
```

I've done a bit of reformatting by adding some newlines but otherwise your output should look like mine.

Study Drills

- Use `UPDATE` to change my name back to "Zed" by my `person.id`.
- Write an `UPDATE` that renames any dead animals to "DECEASED". If you try to say they are "DEAD" it'll fail because SQL will think you mean 'set it to the column named "DEAD"', which isn't what you want.

Updating Complex Data

In the previous exercise you did a simple `UPDATE` that changed just one row. In this exercise you'll use sub-select queries again to update the `pet` table using information from the `person` and `person_pet` tables.

ex10.sql

```
SELECT * FROM pet;

UPDATE pet SET name = 'Zed''s Pet' WHERE id IN (
    SELECT pet.id
    FROM pet, person_pet, person
    WHERE
    person.id = person_pet.person_id AND
    pet.id = person_pet.pet_id AND
    person.first_name = 'Zed'
);

SELECT * FROM pet;
```

This is how you update one table based on information from another table. There're other ways to do the same thing, but this way is the easiest to understand for you right now.

What You Should See

As usual, I use my little `code.sql` to reset my database and then output nicer columns with `sqlite3 -header -column -echo`.

Exercise 10 Session

```
$ sqlite3 mydata.db < code.sql
# ... output cut ...
$ sqlite3 -header -column -echo mydata.db < ex10.sql
SELECT * FROM pet;
id          name        breed       age         dead
----------  ----------  ----------  ----------  ----------
0           Fluffy      Unicorn     1000        0
1           Gigantor    Robot       1           0

```

```
UPDATE pet SET name = "Zed's Pet" WHERE id IN (
    SELECT pet.id
    FROM pet, person_pet, person
    WHERE
    person.id = person_pet.person_id AND
    pet.id = person_pet.pet_id AND
    person.first_name = "Zed"
);

SELECT * FROM pet;
id          name        breed       age         dead
----------  ----------  ----------  ----------  ----------
0           Zed's Pet   Unicorn     1000        0
1           Zed's Pet   Robot       1           0
$
```

Study Drills

- Write an SQL that only renames dead pets I own to "Zed's Dead Pet".
- Go to the SQL As Understood By SQLite page and start reading through the docs for CREATE TABLE, DROP TABLE, INSERT, DELETE, SELECT, and UPDATE.
- Try some interesting things you find in these docs, and take notes on things you don't understand so you can research them more later.

Portability Notes

We have to use sub-select queries to do this because SQLite3 doesn't support the FROM keyword in the SQL language. In other databases you can do traditional joins in your UPDATE just like you would with SELECT.

Replacing Data

I'm going to show you an alternative way to insert data which helps with atomic replacement of rows. You don't necessarily need this too often, but it does help if you're having to replace whole records and don't want to do a more complicated UPDATE without resorting to transactions.

In this situation, I want to replace my record with another guy but keep the unique id. Problem is I'd have to either do a DELETE/INSERT in a transaction to make it atomic, or I'd need to do a full UPDATE.

Another simpler way to do it is to use the REPLACE command, or add it as a modifier to INSERT. Here's some SQL where I first fail to insert a new record, then I use these two forms of REPLACE to do it:

ex11.sql

```
/* This should fail because 0 is already taken. */
INSERT INTO person (id, first_name, last_name, age)
    VALUES (0, 'Frank', 'Smith', 100);

/* We can force it by doing an INSERT OR REPLACE. */
INSERT OR REPLACE INTO person (id, first_name, last_name, age)
    VALUES (0, 'Frank', 'Smith', 100);

SELECT * FROM person;

/* And shorthand for that is just REPLACE. */
REPLACE INTO person (id, first_name, last_name, age)
    VALUES (0, 'Zed', 'Shaw', 37);

/* Now you can see I'm back. */
SELECT * FROM person;
```

What You Should See

In this exercise I'm going to enter these commands in the sqlite3 console rather than run them from the command line:

Exercise 11 Session

```
sqlite> /* This should fail because 0 is already taken. */
sqlite> INSERT INTO person (id, first_name, last_name, age)
```

```
3     ...>     VALUES (0, 'Frank', 'Smith', 100);
4   Error: PRIMARY KEY must be unique
5   sqlite>
6   sqlite> /* We can force it by doing an INSERT OR REPLACE. */
7   sqlite> INSERT OR REPLACE INTO person (id, first_name, last_name, age)
8     ...>     VALUES (0, 'Frank', 'Smith', 100);
9   sqlite>
10  sqlite> SELECT * FROM person;
11  0|Frank|Smith|100
12  sqlite>
13  sqlite> /* And shorthand for that is just REPLACE. */
14  sqlite> REPLACE INTO person (id, first_name, last_name, age)
15    ...>     VALUES (0, 'Zed', 'Shaw', 37);
16  sqlite>
17  sqlite> /* Now you can see I'm back. */
18  sqlite> SELECT * FROM person;
19  0|Zed|Shaw|37
20  sqlite>
21  sqlite>
```

You can see on line 4 that I get an "Error: PRIMARY KEY must be unique" because the record has the id 0. I want to do a combination DELETE/INSERT with a new record, so then I do an `INSERT OR REPLACE` next. After that I put the record back with `REPLACE` which is shorthand for `INSERT OR REPLACE`.

Study Drills

- Go to the SQLite3 INSERT page and look at the other `INSERT OR` options you have.
- Practice `REPLACE` by replacing the Unicorn with a pet Parrot.

Portability Notes

Some databases might not have the `REPLACE` command or even the `INSERT OR REPLACE` syntax. In that case, you'll just have to wait until I show you transactions.

EXERCISE 12

Destroying And Altering Tables

You've already encountered `DROP TABLE` as a way to get rid of a table you've created. I'm going to show you another way to use it and also how to add or remove columns from a table with `ALTER TABLE`.

ex12.sql

```
1  /* Only drop table if it exists. */
2  DROP TABLE IF EXISTS person;
3
4  /* Create again to work with it. */
5  CREATE TABLE person (
6      id INTEGER PRIMARY KEY,
7      first_name TEXT,
8      last_name TEXT,
9      age INTEGER
10 );
11
12 /* Rename the table to peoples. */
13 ALTER TABLE person RENAME TO peoples;
14
15 /* Add a hatred column to peoples. */
16 ALTER TABLE peoples ADD COLUMN hatred INTEGER;
17
18 /* Rename peoples back to person. */
19 ALTER TABLE peoples RENAME TO person;
20
21 .schema person
22
23 /* We don't need that. */
24 DROP TABLE person;
```

I'm doing some fake changes to the tables to demonstrate the commands, but this is everything you can do in SQLite3 with the `ALTER TABLE` and `DROP TABLE` statements. I'll walk through this so you understand what's going on:

ex21.sql:2 Use the `IF EXISTS` modifier and the table will be dropped only if it's already there. This suppresses the error you get when running your `.sql` script on a fresh database that has no tables.

ex21.sql:5 Just recreating the table again to work with it.

ex21.sql:13 Using `ALTER TABLE` to rename it to `peoples`.

ex21.sql:16 Add a new column `hatred` that is an `INTEGER` to the newly renamed table `peoples`.

ex21.sql:19 Rename `peoples` back to `person` because that's a dumb name for a table.

ex21.sql:21 Dump the schema for `person` so you can see it has the new `hatred` column.

ex21.sql:24 Drop the table to clean up after this exercise.

What You Should See

If you run this script it should look something like this:

Exercise 12 Session

```
$ sqlite3 -echo ex12.db < ex12.sql

DROP TABLE IF EXISTS person;

CREATE TABLE person (
    id INTEGER PRIMARY KEY,
    first_name TEXT,
    last_name TEXT,
    age INTEGER
);

ALTER TABLE person RENAME TO peoples;
ALTER TABLE peoples ADD COLUMN hatred INTEGER;
ALTER TABLE peoples RENAME TO person;

.schema person

CREATE TABLE "person" (
    id INTEGER PRIMARY KEY,
    first_name TEXT,
    last_name TEXT,
    age INTEGER
, hatred INTEGER);

DROP TABLE person;
$
```

I've added some extra spacing so you can read it easier, and remember to pass in the `-echo` argument so it prints out what it's run.

Study Drills

- Update your `code.sql` file you've been putting all the code in so that it uses the `DROP TABLE IF EXISTS` syntax.
- Use `ALTER TABLE` to add a `height` and `weight` column to `person` and put that in your `code.sql` file.
- Run your new `code.sql` script to reset your database and you should have no errors.

Portability Notes

`ALTER TABLE` is a collection of everything a database vendor couldn't put into their SQL syntax. Some databases will let you do more with tables than other databases, so read up on the documentation and see what's possible.

Migrating And Evolving Data

This exercise will have you apply some skills you've learned. I'll have you take your database and "evolve" the schema to a different form. You'll need to make sure you know the previous exercise well and have your `code.sql` working as we'll be using it during this exercise. If you don't have either of these then go back and get everything straightened out.

For this exercise I'll show you another way to load your `code.sql` and begin working with the database using the `-init` option to `sqlite3`:

Exercise 13 Session

```
$ sqlite3 -init code.sql ex13.db
-- Loading resources from code.sql
0|Zed|Shaw|37
Fluffy|1000
Gigantor|1
Fluffy|1000
0|Fluffy|1000|0
1|Gigantor|1|1
Gigantor|1
0|Fluffy|Unicorn|1000|0
0|Fluffy|Unicorn|1000|0
1|Gigantor|Robot|1|0

SQLite version 3.13.0 2016-05-18 10:57:30
Enter ".help" for usage hints.
sqlite> .schema
CREATE TABLE person (
    id INTEGER PRIMARY KEY,
    first_name TEXT,
    last_name TEXT,
    age INTEGER
);
CREATE TABLE pet (
    id INTEGER PRIMARY KEY,
    name TEXT,
    breed TEXT,
    age INTEGER,
    dead INTEGER
);
CREATE TABLE person_pet (
```

```
    person_id INTEGER,
    pet_id INTEGER
);
sqlite>
```

Notice how I first load the `code.sql` file and it drops me into the `sqlite3` interactive prompt. This is where you can now work with the database live. I then type `.schema` to make sure I know how the database looks.

The Assignment

What you're tasked with doing is the following list of changes to the database:

- Add a `dead` column to `person` that's like the one in `pet`.
- Add a `phone_number` column to `person`.
- Add a `salary` column to `person` that is `float`.
- Add a `dob` column to both `person` and `pet` that is a `DATETIME`.
- Add a `purchased_on` column to `person_pet` of type `DATETIME`.
- Add a `parent` to `pet` column that's an `INTEGER` and holds the `id` for this pet's parent.
- Update the existing database records with the new column data using `UPDATE` statements. Don't forget about the `purchased_on` column in `person_pet` relation table to indicate when that person bought the pet.
- Add 4 more people and 5 more pets and assign their ownership and what pets are parents. On this last part remember that you get the id of the parent, then set it in the `parent` column.
- Write a query that can find all the names of pets and their owners bought after 2004. Key to this is to map the `person_pet` based on the `purchased_on` column to the `pet` and `parent`.
- Write a query that can find the pets that are children of a given pet. Again look at the `pet.parent` to do this. It's actually easy so don't over think it.

You should do this by writing a `ex13.sql` file with these new things in it. You then test it by resetting the database using `code.sql` and then running `ex13.sql` to alter the database and run the `SELECT` queries that confirm you made the right changes.

Take your time working on this, and when you're done your schema should look like this:

Exercise 13-result Session

```
$ sqlite3 -init ex13.sql ex13.db
-- Loading resources from ex13.sql

SQLite version 3.13.0 2016-05-18 10:57:30
Enter ".help" for usage hints.
sqlite> .schema
CREATE TABLE person (
    id INTEGER PRIMARY KEY,
    first_name TEXT,
    last_name TEXT,
    age INTEGER
, dead INTEGER, phone_number TEXT, salary FLOAT, dob DATETIME);
CREATE TABLE pet (
    id INTEGER PRIMARY KEY,
    name TEXT,
    breed TEXT,
    age INTEGER,
    dead INTEGER
, parent INTEGER);
CREATE TABLE person_pet (
    person_id INTEGER,
    pet_id INTEGER
, purchased_on DATETIME);
sqlite>
```

Watch The Video

In the previous exercises you could probably get fairly far without watching the video, but this exercise is more difficult. In the video you'll be able to see how I would work with a database to evolve it. My process is to load the database in interactive mode like I show you here, then to type SQL commands until I figure out what I want. I copy and paste the commands that work into my `ex13.sql` file. Once I have a basic `ex13.sql` file working I drop out of interactive mode and continue with just running the `ex13.sql` like a script. Watch the video to see me do this and try to follow along if you can't figure out each of these operations. You may have to spend some time on this exercise.

Basic Transactions

Imagine if the SQL in your last exercise had an error half-way through its run and it aborted. You may have even run into this problem, and then you see that your database is now seriously broken. You've been getting away with this because you have a big `code.sql` file that rebuilds your database, but in a real situation you can't trash your whole database when you mess up.

What you need to make your script safer is the `BEGIN`, `COMMIT`, and `ROLLBACK` commands. These start a transaction, which creates a "boundary" around a group of SQL statements so you can abort them if they have an error. You start the transaction with `BEGIN`, do your SQL, and then when everything's good end the transaction with `COMMIT`. If you have an error, then you just issue `ROLLBACK` to abort what you did.

For this exercise I want you to do the following:

- Take your `ex13.sql` and copy it to `ex14.sql` so you can modify it.
- Once you have that, put a `BEGIN` at the top and a `ROLLBACK`. At the bottom.
- Now run it and you'll see that it's as if your script didn't do anything.
- Next, change the `ROLLBACK` to be `COMMIT` and run it again, and you'll see it works like normal.
- Get rid of the `BEGIN` and `COMMIT` from your `ex14.sql` so it's back the way it was.
- Now create an error by removing one of the `TABLE` keywords from one of the lines. This is so you can make it have an error and recover.

Once you have this broken `ex14.sql` you'll play with it in the `sqlite3` console to do a recovery:

Exercise 14 Session

```
$ sqlite3 ex14.db
SQLite version 3.7.8 2011-09-19 14:49:19
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
sqlite>
sqlite> .read code.sql
/* cut for simplicity */

sqlite>
sqlite> BEGIN;
sqlite> .read ex14.sql
Error: near line 5: near "person_pet": syntax error
```

```
14 sqlite> .schema
15 /* cut the schema with partial alters in it */
16 sqlite> ROLLBACK;
17 sqlite>
18 sqlite> .schema
19 CREATE TABLE person (
20     id INTEGER PRIMARY KEY,
21     first_name TEXT,
22     last_name TEXT,
23     age INTEGER
24 );
25 CREATE TABLE person_pet (
26     person_id INTEGER,
27     pet_id INTEGER
28 );
29 CREATE TABLE pet (
30     id INTEGER PRIMARY KEY,
31     name TEXT,
32     breed TEXT,
33     age INTEGER,
34     dead INTEGER
35 );
36 sqlite>
```

This one's long so I'm going to break it down so you can track what's going on:

1 I start up `sqlite3` so I can get into the console with `ex14.db`.

7 I then `.read` the `code.sql` file to setup the database like normal but doing it from within sqlite3.

11 I enter `BEGIN` so I can start a transaction boundary.

12 I run `.read ex14.sql` to run it, but remember it has an error so it aborts. My error was on line 5 but yours could be anywhere else.

14 I then run `.schema` so you can see that some changes actually were made, and I'll want to undo them.

16 Since the last command had an error I run `ROLLBACK` here to abort all the things I did since the `BEGIN` on line 11.

18 To show that the database is back the way it should be, I do another `.schema` and you can see all that junk is now gone.

Study Drills

- Read the instructions on SQLite3 transactions to get an idea of what's possible.
- Use transactions to bound some example UPDATES and INSERTS to see how those work too.
- Try using the alternative syntax of `BEGIN TRANSACTION`, `COMMIT TRANSACTION`, and `ROLLBACK TRANSACTION`.

Portability Notes

Some databases don't have the same rollback and commit semantics as other databases. Some also don't understand the syntax with the word `TRANSACTION`. They should universally abort everything you did, but again look at your manual to confirm this is true.

GROUP BY And Counts

For this exercise we'll need a large dataset to demonstrate how to group rows by data in their columns. Right now you only have a few pets and an owner in your database. What we need is a huge number of pets and owners. What I've done is created a dump of a fake dataset that you can load.

Once you have that file, you simply load it into your database using `-init` and you can start playing with the `GROUP BY` style of query.

What Is `GROUP BY`

The best way to describe `GROUP BY` is by comparing it to what a sociologist would need to analyze a dataset. A sociologist isn't necessarily interested in what any single row contains. A sociologist doesn't care about each person's eye color, height, or survey answers. What matters most is the *summary* of those variables (columns) into numeric groups. These groups might be:

- Counts of how many people answered "yes" vs. "no" to a question.
- Average height of people by geographic region.
- Counts of each gender and who they voted for in an election.

A sociologist wants the database to go through all the rows, and calculate a *summary* number based on separate *groups* of rows. Another way to put this is the sociologist wants to have the rows separated out into buckets by one column, and then the contents of those buckets counted or averaged. This is what `GROUP BY` does for the sociologist.

In your database of pets we want to calculate some counts of these groups to look for patterns just like a sociologist would. We might want to know the following:

- How many pets are dead vs. alive?
- At what ages do pets die?
- How many people own that are alive vs. dead?

We can also start calculating averages, but in this exercise we'll focus on basic counts and groups to understand this concept.

Generating Fake Data

To get started I created a simple little Python script that will output a `ex15.sql` file with fake person and pet data in it. I have included a file `LSQL_Extra_Files.zip` that contains this code, an `ex15.sql` file it generated, and the `code/words.txt` file you need to run it. You can either run the `ex15.py` file yourself and make a new `ex15.sql` or just use the `ex15.sql` I generated. Do not copy-paste this code though as the quote characters (”, ’) will not be correct.

ex15.py

```
import random
from datetime import timedelta, datetime

words = [x.strip().capitalize() for x in open("code/words.txt").readlines()]
breeds = ['Dog', 'Cat', 'Fish', 'Unicorn']

pet_names = words[:len(words)/2]
owner_names = words[len(words)/2:]

owner_insert = '''INSERT INTO person
    VALUES(NULL, '{first}', '{last}', '{age}', '{dob}');'''
pet_insert = '''INSERT INTO pet
    VALUES(NULL, '{name}', '{breed}', '{age}', '{dead}');'''
relate_insert = '''INSERT INTO person_pet
    VALUES({person_id}, {pet_id}, '{purchased_on}');'''

sql = []

def random_date():
    years = 365 * random.randint(0, 50)
    days = random.randint(0, 365)
    hours = random.randint(0, 24)
    past = timedelta(days=years + days, hours=hours)
    return datetime.now() - past

# make a bunch of random pets
for i in range(0, 100):
    age = random.randint(0, 20)
    breed = random.choice(breeds)
    name = random.choice(pet_names)
    dead = random.randint(0,1)
    sql.append(pet_insert.format(
        age=age, breed=breed, name=name, dead=dead))

```

```
# make a bunch of random people
for i in range(0, 100):
    age = random.randint(0, 100)
    first = random.choice(owner_names)
    last = random.choice(owner_names)
    dob = random_date()
    sql.append(owner_insert.format(
        age=age, first=first, last=last, dob=dob))

# only sample from the middle of IDs to connect pets and people
# this avoids the first few IDs that have been taken in the db
# and ensures some unowned pets and owners without pets
for i in range(0,75):
    person_id = random.randint(5,95)
    pet_id = random.randint(5,95)
    purchased_on = random_date()
    sql.append(relate_insert.format(
        person_id=person_id, pet_id=pet_id, purchased_on=purchased_on))

# this is descructive, normally you'd check and ask to overwrite
with open("ex15.sql", "w") as output:
    output.write("\n".join(sql))
```

Loading The Data

To load the data you'll need to run these shell commands:

Exercise 15 Session

```
$ rm -rf ex15.db
$ python ex15.py
$ ls -l ex15.sql
-rw-r--r--  1 zedshaw  staff  14899 Oct 14 09:53 ex15.sql
$ sqlite3 ex15.db < code.sql
# ... cut output of the code.sql run
$ sqlite3 ex15.db < ex15.sql
$ sqlite3 ex15.db
```

What You Should See

Using the `-init` option is fine when you're loading a database for the first time, but that rarely happens when you work with databases. It's also annoying to work with data from `.sql` files only. A better way is to interact with the data, interrogating it and studying it, until you develop a nice set of queries you can work with in a `.sql` file. In this exercise I'm going to do the following interactive session, but you should watch the video to really get the feel for how I work with data.

Here's a sample session of me interacting with it:

Exercise 15 Session

```
sqlite> /* how many dead pets are there */
sqlite> select dead, count(*) from pet group by dead;
0|42
1|60
sqlite> /* how many per each breed */
sqlite> select breed, count(*) from pet group by breed;
Cat|27
Dog|24
Fish|24
Robot|1
Unicorn|26
sqlite> /* how many dead per breed */
sqlite> select breed, dead from pet group by breed;
Cat|1
Dog|0
Fish|1
Robot|0
Unicorn|1
sqlite> /* not quite right, add a count */
sqlite> select breed, dead, count(dead) from pet group by breed;
Cat|1|27
Dog|0|24
Fish|1|24
Robot|0|1
Unicorn|1|26
sqlite> /* still not right, need dead/alive count */
sqlite> select breed, dead, count(dead) from pet group by breed, dead;
Cat|0|8
Cat|1|19
Dog|0|12
Dog|1|12
Fish|0|11
```

```
Fish|1|13
Robot|0|1
Unicorn|0|10
Unicorn|1|16
sqlite>
```

Study Drills

GROUP BY With Relations

For the rest of the book we will be using the `ex15.db` file but you should rename it to `thw.db` to make it easier to work with from now on. Use the `mv` command in your terminal to do this:

```
mv ex15.db thw.db
```

We'll now explore how to do a `GROUP BY` but using the relationships between the `pet`, `person`, and `person_pet` tables. Grouping together data in one table isn't nearly as useful as doing it with the results of multiple tables together. It's the relationships *between* tables that is usually the most interesting thing about databases.

Doing a `GROUP BY` on multiple tables is easy once you realize that `SELECT` simply returns a new table. When you run `SELECT` on `pet`, `person`, and `person_pet` you'll link them together, and the result is a giant table. You then just have to append a `GROUP BY` that summarizes the data in this new table however you want. That means a good process for figuring out a `GROUP BY` is the following:

1. Work out a correct `SELECT` query that gives you a new table of data you want to summarize.
2. Add `count()` or other summary functions to the list of returned columns, and remove any that aren't needed for the summary.
3. Append the `GROUP BY` clause at the end describing what columns should be used to group this table together.

Here's the SQL demonstrating this process using the Exercise 15 example to analyze only the pets that have owners:

ex16.sql

```
/* simple query to get a related table */
select * from person, person_pet, pet
    where person.id = person_pet.person_id and pet.id = person_pet.pet_id;

/* add a basic count column and append the GROUP BY */
select person.first_name, pet.breed, pet.dead, count(dead)
    from person, person_pet, pet
    where person.id = person_pet.person_id and pet.id = person_pet.pet_id
    group by pet.breed, pet.dead;

/* drop the person.first_name since that's not summarized */
select pet.breed, pet.dead, count(dead)
    from person, person_pet, pet
```

```
    where person.id = person_pet.person_id and pet.id = person_pet.pet_id
    group by pet.breed, pet.dead;

/* Compare the counts to without the person_pet relation */
select breed, dead, count(dead)
    from pet
    group by breed, dead;
```

What You Should See

Before you follow along, make sure that you have the database `thw.db` that you built in Exercise 15 (remember that we copied it from `ex15.db` in this exercise). Once you have that you can try it out like this:

Exercise 16 Session

```
/* simple query to get a related table */
sqlite> select * from person, person_pet, pet where person.id = person_pet.person_id and pe
0|Zed|Shaw|37|0|0|0|Fluffy|Unicorn|1000|0
0|Zed|Shaw|37|0|1|1|Gigantor|Robot|1|0
...

/* add a basic count column and append the GROUP BY */
sqlite> select person.first_name, pet.breed, pet.dead, count(dead) from person, person_pet,
Coal|Cat|0|5
Direction|Cat|1|17
...

/* drop the person.first_name since that's not summarized */
sqlite> select pet.breed, pet.dead, count(dead) from person, person_pet, pet where person.i
Cat|0|5
Cat|1|17
Dog|0|9
Dog|1|9
Fish|0|9
Fish|1|6
Robot|0|1
Unicorn|0|10
Unicorn|1|11

/* Compare the counts to without the person_pet relation */
sqlite> select breed, dead, count(dead) from pet group by breed, dead;
Cat|0|8
```

```
Cat|1|19
Dog|0|12
Dog|1|12
Fish|0|11
Fish|1|13
Robot|0|1
Unicorn|0|10
Unicorn|1|16
```

You'll see that I start by doing just a basic big data dump query to make sure I have everything I need. Once I have that, I evolve the query in steps to make sure that it comes out correctly. At the end I compare this final query to the original `SELECT` in Exercise 15, confirming that the counts are smaller. This makes sense since there're fewer pets that are owned than pets that aren't. You might need to scan through the full table and possibly craft a smaller database if you want to dig deep and figure out what's going on. Watch the video for this exercise to see me do this.

Study Drills

- Try a `GROUP BY` that is only between two tables in every combination you can imagine.
- Try a `GROUP BY` that uses a sub-select query instead.

Dates, Times and ORDER BY

I'm going to tell you a little story about SQL and time. I'm not sure if this story is exactly true, but it's what I was told when I asked why dates are so totally weird in SQL. Back in the early days of computers the main thing that made money was big giant computers, and big giant databases. Mostly because the government and banks needed them. During this time there were huge wars between giant computer companies and to create peace among them the SQL standard was born. With the standard in place these companies could sell their databases to customers who didn't need to worry if their Database Administrators (DBA) would need to be retrained.

The problem is, if every database uses the exact same language then they become a commodity. IBM's database too expensive? With truly standardized SQL you can just buy an Oracle database and flip a switch. Software was (and still is) very much about locking people into a deadly embrace to extract long term profits for substandard goods. To prevent this commodity status, the SQL companies looked for holes in the standard that they could tweak so that *their* version of SQL was "compliant" with the standard, but also had something very specific to their database that prevented you from changing to a competitor.

The perfect match was found in dates and time. You see, calendar systems in software are a total disastrous mess already with no real logical standards. Rather than try to standardize time itself, the SQL standard simply left it open ended and terribly unspecified. Through this tiny doorway the database companies crammed every possible little differentiation they could to keep customers from jumping ship. Every database stores time different, formats it weirdly, and has their own flavor because of the accident of history that it kept banks from switching to Oracle from IBM.

Whether that story is true or not, it is true that dates in SQL are *always* weird and you *always* have to look them up for every database you touch. Thankfully most of the database access you'll do will be through abstraction layers that take your fancy Python or Ruby code and translate it into the flavor of SQL your database needs. But, when you're doing manual SQL work, you have to look it up. Don't make any assumptions about how the dates are stored, accessed, formatted, or even *if* they have a date column type.

What?! Yes, believe it or not SQLite3 does not have an actual date type. It just writes your date into a text type and lets you deal with the dates using a set of functions. The good news is that SQLite3 has really great date and time functions that handle many of the calendar calculations you could ever need. Read the Date and Time Functions documentation, and then try these sample queries:

ex17.sql

```
1  /* everyone who is older than 10 */
2  select * from person where dob < date('now', '-10 years') order by dob;
3
4  /* everyone born before 1970 */
```

```
select * from person where dob < date('1970-01-01') order by dob;

/* all pets purchased this year */
select * from person_pet
    where purchased_on > date('now', 'start of year')
    order by purchased_on;

/* all pets purchased between 1990 and 2000 */
select * from person_pet
    where purchased_on > date('1990-01-01') and purchased_on < date('2000-01-01')
    order by purchased_on;

/* link the pets from the last query */
select pet.name, pet.breed, pet.age, pet.dead, person_pet.purchased_on
    from pet, person_pet
    where
    purchased_on > date('1990-01-01') and
    purchased_on < date('2000-01-01') and
    person_pet.pet_id = pet.id
    order by purchased_on, pet.age;
```

You'll see in this code we are using the dates to get different results, but we're *also* using an `ORDER BY` clause at the end to sort them by the dates. The `ORDER BY` clause works similar to `GROUP BY` and specifies how to sort the records, in order of importance. In the last query we are saying sort the results by `person_pet.purchased_on` and *then* by `pet.age`.

What You Should See

Exercise 17 Session

```
sqlite> select * from person where dob < date('now', '-10 years');
1|Deer|Death|63|1971-01-20 08:23:21.082947
2|Cat|Dinosaurs|56|1992-06-22 06:23:21.083348
....

sqlite> select * from person_pet where purchased_on > date('now', 'start of year');
0|1|2016-10-19

sqlite> select * from person_pet where purchased_on > date('1990-01-01') and purchased_on <
15|65|1990-11-07 20:23:21.085485
38|69|1992-03-11 18:23:21.085703
...
```

```
13
14 sqlite> select pet.name, pet.breed, pet.age, pet.dead from
15    ...>     pet, person_pet where
16    ...>     purchased_on > date('1990-01-01') and
17    ...>     purchased_on < date('2000-01-01') and
18    ...>     person_pet.pet_id = pet.id;
19 Camera|Fish|0|1
20 Aftermath|Cat|10|1
21 ...
```

Study Drills

- Combine the `GROUP BY` with these date queries to see if you can produce counts over various time periods.
- Go back and add `ORDER BY` to queries from other exercises to see how it works.

Portability Notes

As mentioned in the introduction to this exercise, dates are a portability nightmare. Always consult the documentation before using dates and times.

Aggregate Functions

In this exercise you'll learn the other "aggregate functions". You've already played with `count()`, so these should be easy as they are used the same.

avg()

The `avg()` function works exactly like `count()` does except that it will do a simple average over the column. It is most useful when it's combined with `GROUP BY` to do summary statistics on different categories of rows. In this exercise I'm going to give you just a simple sample of how `avg()` works. Then you will study the documentation for `avg()` and attempt to combine it with what you know so far to produce more interesting queries.

ex18.sql

```
/* select the average age of every person */
select avg(age) from person;

/* select the average age of every pet */
select avg(age) from pet;

/* select the average age of every breed by dead or alive */
select breed, dead, avg(age) from pet group by breed, dead;
```

sum() and total()

You can use `sum()` or `total()` to calculate the sum of column. Keep in mind that this is different from `count()` which only counts the number of *rows* in a query. The `sum()` will add up the *contents* of a column and return the sum/total.

ex18.sql

```
select sum(age) from person;

select sum(age) from pet;
```

```
4
5  select breed, sum(dead), sum(age) from pet where dead = 1 group by breed, dead;
```

min() and max()

You can find the smallest value (`min()`) and largest values (`max()`) in the query.

ex18.sql

```
1  select min(age), max(age) from person;
2
3  select min(age), max(age) from pet;
4
5  select breed, dead, min(age), max(age) from pet
6      where age > 0
7      group by breed, dead;
```

What You Should See

Exercise 18 Session

```
1  sqlite> select avg(age) from person;
2  50.990099009901
3  sqlite> select avg(age) from pet;
4  18.4313725490196
5  sqlite> select breed, dead, avg(age) from pet group by breed, dead;
6  Cat|0|10.0
7  Cat|1|9.22222222222222
8  Dog|0|8.66666666666667
9  Dog|1|6.28571428571429
10 Fish|0|6.25
11 Fish|1|10.0555555555556
12 Robot|0|1.0
13 Unicorn|0|85.6153846153846
14 Unicorn|1|6.66666666666667
15 sqlite> select sum(age) from person;
16 5150
17 sqlite> select sum(age) from pet;
18 1880
19 sqlite> select breed, sum(dead), sum(age) from pet where dead = 1 group by breed, dead;
```

```
Cat|18|166
Dog|7|44
Fish|18|181
Unicorn|12|80
sqlite> select min(age), max(age) from person;
4|100
sqlite> select min(age), max(age) from pet;
0|1000
sqlite> select breed, dead, min(age), max(age) from pet
   ...>     where age > 0
   ...>     group by breed, dead;
Cat|0|4|20
Cat|1|1|19
Dog|0|2|15
Dog|1|1|14
Fish|0|3|9
Fish|1|2|19
Robot|0|1|1
Unicorn|0|3|1000
Unicorn|1|1|13
```

Study Drills

- Study the SQLite3 Aggregate Functions documentation. It's very thin, so you will want to research more places on the internet to figure out how to use it.
- Use the `avg()` combined with `GROUP BY`, `count()`, and linking `pet`, `person`, and `person_pet` together to produce interesting analysis of the data. Remember that you can do this by first working out each component of the query, then combining it. Watch the video *after* attempting it on your own to see how I do this.
- Do the same for `sum()`, `min()`, and `max()` using this data.

Portability Notes

Not every database will have this, and if they do they may use it differently. Always read the documentation for your database before you use functions like `avg()`.

Logic and Math Expressions

The SQLite3 language supports many more functions to handle math, string manipulation, grouping, types, and other features in the SQLite3 Core Functions documentation. The only catch is sometimes you'll want to use the AS keyword to assign these results to a fake column name. It's best to always give math expressions a column name with AS from now on, but if you ever type out an expression and don't see it, that's why. To use AS just do this:

```
select count(*) as rowcount from pet;
select avg(age) as average from pet;
```

Those two examples simply give the computation to perform (`count(*)` or `avg(age)`) and then creates a fake column name for it (`rowcount` or `average`).

Here's a few sample expressions to show you what you can do:

ex19.sql

```
1  /* get a simple average for pet ages */
2  select sum(age) / count(*) as average from pet;
3
4  /* compare it to the avg() function */
5  select avg(age) from pet;
6
7  /* get the average name length (anl) of pets */
8  select avg(length(name)) as anl from pet;
9
10 /* get the avg() age rounded */
11 select round(avg(age)) as average from pet;
12
13 /* get a random number */
14 select random();
15
16 /* use modulus and abs() to make random 0-20 */
17 select abs(random() % 20);
18
19 /* use update to change all pet ages to random 0-20 */
20 update pet set age = abs(random() % 20) where dead = 0;
21
22 /* check that it changed (maybe) */
23 select round(avg(age)) as average from pet;
24
25 /* use 0-50 as the range */
```

```
update pet set age = abs(random() % 50) where dead = 0;

/* check the average again */
select round(avg(age)) as average from pet;
```

These are fairly easy to understand, except the code:

```
update pet set age = abs(random() % 20) where dead = 0;
```

This seems weird because you'd think that this would use `random()` to get *one* value, then assign that to *all* of the rows for the pet age. However, SQL is smart enough to know you mean to run this `abs(random() % 20)` expression once for each row, changing them all. This is a handy trick if you need to modify all the rows by a certain amount to correct errors. It is also a quick way to completely trash your database so be careful when running `UPDATE` with math expressions.

What You Should See

Keep in mind that *your* numbers will be different because you are starting out with random data and then making more random data.

Exercise 19 Session

```
sqlite> select sum(age) / count(*) as average from pet;
10
sqlite> select avg(age) from pet;
10.3235294117647
sqlite> select avg(length(name)) as anl from pet;
5.81372549019608
sqlite> select round(avg(age)) as average from pet;
10.0
sqlite> select random();
-115791249973628983
sqlite> select abs(random() % 20);
6
sqlite> update pet set age = abs(random() % 20) where dead = 0;
sqlite> select round(avg(age)) as average from pet;
9.0
sqlite> update pet set age = abs(random() % 50) where dead = 0;
sqlite> select round(avg(age)) as average from pet;
16.0
```

Study Drills

- Play around with the random age generation `UPDATE` statements and the check for the average with `round(avg(age))`. You may be surprised that no matter how many times you run an update you get the same average when rounded. This is because the random function used in SQLite3 is not very good and only produces barely random numbers.
- Go through all the other expressions listed on that page and try them out with your `thw.db` database file.

Portability Notes

These functions may or may not be available on different databases and the most certainly will work differently. As usual, never assume that one database implements a function the same as any other database. Maybe MySQL does a better `random()` function, or another database may not even have one. Always read the documentation.

Inner and Outer Joins

You have been learning how to join (link) tables using simple equality (`person.id = person_pet.person_id`), and sub-select queries. There is one more way to link them with `INNER JOIN` and `OUTER JOIN`. I personally don't use these two types of joins because I *always* have to look up what they do and why they exist. They seem like some odd feature that was added to SQL because one of the companies selling databases added it to their server and that meant everyone else had to have it too. The phrases "inner" and "outer" also make no sense in the context of tables. They are just arbitrary words that you have to memorize to use them, rather than a description of what they actually do. I'm sure someone smarter than me will be able to explain the reason these words were chosen but who cares. They are weird and not very useful to be brutally honest.

This exercise is mostly so you'll know what these are when you see them, and you'll know to avoid these kinds of joins without first doing some serious research. Let's start this discussion with three example queries on our database:

ex20.sql

```
select * from pet, person, person_pet
    where person.id = person_pet.person_id
    and pet.id = person_pet.pet_id
    and person.first_name = 'Zed';

select * from pet join person, person_pet
    on person.id = person_pet.person_id
    and pet.id = person_pet.pet_id
    and person.first_name = 'Zed';

select * from pet left outer join person, person_pet
    on person.id = person_pet.person_id
    and pet.id = person_pet.pet_id
    and person.first_name = 'Zed';
```

Run these queries and analyze the output in the WYSS section where I'll discuss what's going on.

What You Should See

When you run these, you'll see they produce the exact same output:

Exercise 20 Session

```
sqlite> select * from pet, person, person_pet
   ...>     where person.id = person_pet.person_id
   ...>     and pet.id = person_pet.pet_id
   ...>     and person.first_name = "Zed";
0|Fluffy|Unicorn|6|0|0|Zed|Shaw|37||0|0|
1|Gigantor|Robot|45|0|0|Zed|Shaw|37||0|1|2016-10-19
sqlite>
sqlite> select * from pet join person, person_pet
   ...>     on person.id = person_pet.person_id
   ...>     and pet.id = person_pet.pet_id
   ...>     and person.first_name = "Zed";
0|Fluffy|Unicorn|6|0|0|Zed|Shaw|37||0|0|
1|Gigantor|Robot|45|0|0|Zed|Shaw|37||0|1|2016-10-19
sqlite>
sqlite> select * from pet left outer join person, person_pet
   ...>     on person.id = person_pet.person_id
   ...>     and pet.id = person_pet.pet_id
   ...>     and person.first_name = "Zed";
0|Fluffy|Unicorn|6|0|0|Zed|Shaw|37||0|0|
1|Gigantor|Robot|45|0|0|Zed|Shaw|37||0|1|2016-10-19
sqlite>
```

In fact, the only difference between the first two is I replaced one , (comma) with the word `JOIN` and the word `WHERE` with the word `ON`. In the third query I've done nearly the same thing except I added the word `OUTER` before join. The third query *should* produce different results, but that seems to only work in various situations involving missing columns of data between the three tables. An outer join is supposed to fill in `null` values where there is missing data, but it depends on whether you are doing a `LEFT OUTER JOIN` or a `RIGHT OUTER JOIN` and then it's still obtuse as to which one does what.

Like I said, I'm sure somebody smart than I'll ever be could figure out the reasons for these, but if I can do the exact same thing 95% of the time with only simple equality, why bother? I even think many databases simply translate one style of query into the other styles of queries, thus justifying the syntax similarities.

Don't Bother with USING

In many cases the using an `INNER JOIN` or `OUTER JOIN` just requires more syntax to produce the same results. In other situations it ends up being less exact or requires specific database designs that are just not feasible in modern systems. For example, if you want to use the `USING` syntax like this:

```
select * from pet inner join person, person_pet using (id);
```

You'll get an error because the three tables (`pet`, `person_pet`, `person`) because "column id not present in both tables". What? All three tables have an `id` column, so we have two problems here. First, `USING` expects your tables to share a common field. Modern databases are usually auto-generated from ORM systems that do...whatever they want. If the ORM is written by a programmer who knows about `USING`, then it will share the IDs necessary. Otherwise, you'll have different fields for ID fields than found in relation tables.

Further Study

At this point in the design of SQL you hit the "architecture astronaut" features. If you want to be proficient and pass a SQL job interview then you should know these like the back of your hand. The "Next Steps" chapter at the end of this book will give you more than enough books to study for this purpose. If you just want to be able to do basic SQL to understand what's going on, then knowing about these are enough.

Portability Notes

Be prepared for strange results in speed and performance in each database if you use some kinds of joins.

Using Views

A `VIEW` is a fake table that is populated with the results of a `SELECT` query you write. Since every `SELECT` returns a table, it makes sense that you could make those tables permanent. Constructing a `VIEW` is similar to a combination of the `CREATE TABLE` statement and the sub-`SELECT` statement you just learned. If you find yourself frequently using a sub-select or the same query repeatedly, then consider making it a view so you can use the data better.

Here's an example that uses a query from Exercise 16:

ex21.sql

```
/* test our query from Ex16 */
select pet.breed, pet.dead, count(dead)
    from person, person_pet, pet
    where person.id = person_pet.person_id and pet.id = person_pet.pet_id
    group by pet.breed, pet.dead;

/* create the view */
create view dead_pets as
select pet.breed, pet.dead, count(dead) as total
    from person, person_pet, pet
    where person.id = person_pet.person_id and pet.id = person_pet.pet_id
    group by pet.breed, pet.dead;

/* try it */
select * from dead_pets where total > 10;

/* get rid of the Cats to see if it changes dead_pets */
delete from pet where breed = 'Cat';

/* see? it worked */
select * from dead_pets;

/* drop it */
drop view dead_pets;
```

You should notice that I added an `as total` column for the `count(dead)` function. This lets me query against that column even though it is calculated.

What You Should See

We now have a dead_pets table to use:

Exercise 21 Session

```
sqlite> select pet.breed, pet.dead, count(dead) as total
   ...>     from person, person_pet, pet
   ...>     where person.id = person_pet.person_id and pet.id = person_pet.pet_id
   ...>     group by pet.breed, pet.dead;
Dog|0|14
Dog|1|3
Fish|0|6
Fish|1|7
Robot|0|1
Unicorn|0|10
Unicorn|1|10
sqlite> create view dead_pets as
   ...> select pet.breed, pet.dead, count(dead) as total
   ...>     from person, person_pet, pet
   ...>     where person.id = person_pet.person_id and pet.id = person_pet.pet_id
   ...>     group by pet.breed, pet.dead;
sqlite>
sqlite> select * from dead_pets where total > 10;
Dog|0|14
sqlite> delete from pet where breed = 'Cat';
sqlite> select * from dead_pets;
Dog|0|14
Dog|1|3
Fish|0|6
Fish|1|7
Robot|0|1
Unicorn|0|10
Unicorn|1|10
sqlite> drop view dead_pets;
```

Study Drills

- Read about views in the SQLite3 Views Documentation to find out what they can and cannot do.
- Try other queries you've made in the book.

Portability Notes

This is a standard feature of SQL, but there are different restrictions for different databases. Read the documentation to see what each database allows.

Accessing SQLite3 from Python

You can easily access SQLite3 from Python using the DB-API 2.0 interface. Here is a very simple example using the queries from Exercise 16:

ex22.py

```
import sqlite3
conn = sqlite3.connect('thw.db')

cursor = conn.cursor()

cursor.execute("""
select * from person, person_pet, pet
    where person.id = person_pet.person_id and pet.id = person_pet.pet_id;
""")

for row in cursor:
    print row

conn.close()
```

The general pattern for using the Python library is to:

1. Create a connection to the database using its file name.
2. Get a cursor from that connection to do work.
3. Do the work with the cursor using any of the CRUD operations.
4. Call `conn.commit()` when you are ready to commit that work. This is only necessary if you change the database.
5. Close with `conn.close()` when you are done with the database.

Most other languages follow this pattern, but these days most people interact with databases using some kind of Object-Relational Mapping (ORM), but that is outside the scope of this little book.

What You Should See

When you run this code you'll see it print out all the rows from the database as tuples, and each cell of the tuple will be a column. Here's a small snippet of my output:

ex22_output.py

```
(0, u'Zed', u'Shaw', 37, None, 0, 0, None,
    0, u'Fluffy', u'Unicorn', 6, 0)

(0, u'Zed', u'Shaw', 37, None, 0, 1,
    u'2016-10-19', 1, u'Gigantor', u'Robot', 45, 0)

(38, u'Death', u'Doll', 4, u'1972-07-19 16:23:21.084108',
    38, 21, u'1978-01-17 20:23:21.085427', 21,
    u'Aftermath', u'Dog', 1, 1)

(15, u'Crow', u'Duck', 61, u'1991-11-10 19:23:21.083628',
    15, 65, u'1990-11-07 20:23:21.085485',
    65, u'Camera', u'Fish', 0, 1)
```

I've formatted these four rows so they fit on the page and you can read them more easily.

Study Drills

- See if you can map the output from the script I've given to dictionaries instead of tuples.
- If you don't know Python, go find your programming language's SQLite3 library and try to use it.

Portability Notes

While every language will let you do something *like* this script, they will require a different order of operations. Some require explicit transactions and commits. Others aren't very good with threads or reusing connections. They are all going to require research and you may be better off using another library that makes things easier.

Setting Up PostgreSQL

The final exercise in this book will teach you to setup a basic PostgreSQL database on your desktop machine. This will not teach you how to properly manage a PostgreSQL database. It is only going to show you enough to work with one for development purposes and how to load your SQlite3 database into a PostgreSQL database. I am mostly writing this exercise because I *constantly* forget how to setup PostgreSQL on the different machines I need to use. I figure if I'm forgetting how every time then it must be something that other people will definitely need.

This exercise is not a written exercise because installing software is easier to demonstrate in a video than in writing. Please watch the video for your given platform and take notes. There is a video for Linux, OSX, and Windows that you can watch.

WARNING! I may change this after I record the video and write out the steps I took for reference later.

Study Drills

- Using what you've learned from the videos, try to go through all of these exercises again using your new PostgreSQL database setup. You'll find it *mostly* works but you'll have to work around a few syntax issues. This is your first step into how supposedly standard SQL is not very standard.

Next Steps

You now know the *very* basic level of SQL. I wouldn't presume to say you are competent in SQL at all. I'd say you have learned enough to *now* go read another book about SQL. You could stop here and probably do just fine tinkering with databases or investigating why they are broken. That's true. But if you *really* want to become a SQL expert, then I suggest you read the following books:

- Joe Celko's `SQL For Smarties` is massive, but it is a complete course in everything SQL. You could probably read just this book and learn all you need.
- Joe Celko's `Trees and Hierarchies in SQL` is another excellent book if you ever have to make a tree data structure in a SQL database. It's also just a good book for expanding your brain in the realm of SQL.

WARNING! I'll be expanding this section with more books and adding links in the near future.
