1. SQLite is an open source SQL database engine that is the most widely used in the world. This is due to the fact that it is extremely lightweight (isn't resource or disk space hungry) and self contained making it optimal for embedded platforms or with applications that need to run across many platforms. When stating that it is self contained, it means that it requires virtually no support from external libraries or operating systems.

In addition to being self contained, it is also serverless, unlike many other SQL databases. This means that there is no separate server process that must be accessed in order to use the program. This allows SQLite to read and write directly to the disk.

Another feature that makes SQLite so popular is a lack of necessary configuration. You can run the program as-is, directly after downloading without the need to set up users or configuration files or any of the like,

Lastly, SQLite is fully atomic. This is because it is a transactional database, meaning that every transaction either works or fails. If it works, the changes will be successfully fully committed or completely fail and nothing will be modified. Being atomic ensures that your data will not become corrupted or inconsistent, hence the meaning of ACID (Atomic, Consistent, Isolated, and Durable).

2. The SQL language contains many operations, some of the most notable Data Manipulation Language, or DML statements include SELECT, INSERT, DELETE, and UPDATE. The DML I s used to retrieve, store, modify, insert, update, or delete data from a table.

On the other side of the database is the DDL, or Data Definition Language. This is used to modify the database and database objects. Examples of this include the CREATE TABLE, DROP, and ALTER statements.

With the CREATE TABLE statement you can create a new table within a database that follows the schema provided. For example, if we want to create a table of books with the attributes *id*, *title*, and *author*, we would enter the following SQL command into the command line interface of SQLite:

```
CREATE TABLE Books (
id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
title VARCHAR(100) NOT NULL,
author VARCHAR(50) NOT NULL);
```

If we check the schema by running the .schema command, we will get the following:

```
CREATE TABLE Books (
id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
title VARCHAR(100) NOT NULL,
author VARCHAR(50) NOT NULL);
```

At this point there is an empty table named books in the database, if you wanted to add a book to the book table you would use the INSERT statement. To begin, I will add one book from the lab example, and then a few other books I am currently reading by typing the following commands into the SQLite command line window:

```
INSERT into Books (title, author) values("The Elements of
      Statistical Learning", "Trevor Hastie");

INSERT into Books (title, author) values("Darkness Falls", "Kyle
      Mills");

INSERT into Books (title, author) values("iCon", "Jeffrey
      Young");

INSERT into Books (title, author) values("iWoz: Coputer Geek to
      Cult Icon", "Steve Wozniak");
```

At this point all four books will be in the Books table. To verify this, I ran the SELECT statement on all items in the Books table.

```
SELECT * from Books ;

1|The Elements of Statistical Learning|Trevor Hastie

2|Darkness Falls|Kyle Mills

3|iCon|Jeffrey Young

4|iWoz: Coputer Geek to Cult Icon|Steve Wozniak
```

We can see that all of the books were successfully added, but I just noticed a typo in the name of one of the books, specifically ID number 4. To fix this we can update that record with the correct value in the following manner:

```
UPDATE Books SET title=”iWoz: Computer Geek to Cult Icon" WHERE
      id=”4”;
```

Now, if we select the book with ID=4, we can verify that we did indeed fix the typo in the book title. Running the following SQL command:

```
SELECT * from Books where ID=4;
```

returns the now correct result of:

```
4|iWoz: Computer Geek to Cult Icon|Steve Wozniak
```

To find a book with the word “Elements” in the title, we could use something similar to the following:

```
SELECT * from Books where title like "%Elements%";
```

and it returns the following result:

```
1|The Elements of Statistical Learning|Trevor Hastie
```

If we look for a book with the word “Mining” in the title the SELECT statement would look like the following

```
SELECT * from Books where title like "%Mining%";
```

but it returns nothing, because no book has a title that contains the word “Mining”. The first book in the collection should contain this word, so we can update that book to contain the full title, instead of the abbreviated title it has now. To do this we can use the following SQL code:

```
UPDATE Books
...> SET title="The Elements of Statistical Learning: Data
      Mining, Inference, and Prediction"
...> WHERE ID=1;
```

Now, if we run the following command again:

```
SELECT * from Books where title like "%Mining%";
```

it will return:

```
1|The Elements of Statistical Learning: Data Mining, Inference,
and Prediction|Trevor Hastie
```

3. To open SQLite, open a new terminal window and type SQLite3 followed by the name of an existing database, or the name of a new database. You will see the following prompt then:

```
sqlite3 library
SQLite version 3.7.13 2012-06-11 02:05:22
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite>
```

If you are unsure of a command or feature of SQLite, type “.help” into the prompt to display a list of SQLite options you can run in addition to the query commands supported by SQL.

For this example I will create three tables, one holds the users information, one holds the users phone numbers, and the other contains screen names. I used the following SQL to do this:

```
CREATE TABLE User (
id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
first_name VARCHAR(50) NOT NULL,
last_name VARCHAR(50) NOT NULL,
email VARCHAR(100) NOT NULL);

CREATE TABLE UserPhone (
id INTEGER NOT NULL,
pnumber INTEGER NOT NULL,
ptype VARCHAR(25) NOT NULL,
FOREIGN KEY(id) REFERENCES User(id));

CREATE TABLE UserScreenName (
id INTEGER NOT NULL,
sname VARCHAR(50),
imservice VARCHAR(100)
FOREIGN KEY(id) REFERENCES User(id));
```

After creating these tables and reviewing my structure, I have decided that screen names are outdated and no one uses them any more. To remedy this, I will drop the entire UserScreenName table from the database using the following code:

```
DROP TABLE IF EXISTS UserScreenName;
```

To make sure the table was successfully dropped we can type .schema to view the current schema and make sure it looks correct. It appears as everything worked:

```
.schema
CREATE TABLE User (
id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
first_name VARCHAR(50) NOT NULL,
last_name VARCHAR(50) NOT NULL,
email VARCHAR(100) NOT NULL);
CREATE TABLE UserPhone (
id INTEGER NOT NULL,
pnumber INTEGER NOT NULL,
ptype VARCHAR(25) NOT NULL,
FOREIGN KEY(id) REFERENCES User(id));
```

I have also decided that I would like to collect the users age and store it in the User table, so I will alter the schema by running the following command:

```
ALTER TABLE User
ADD age INTEGER;
```

This will add a column with the attribute name age and a type of integer to the end of the database table. Now the schema looks like the following:

```
CREATE TABLE User (
id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
first_name VARCHAR(50) NOT NULL,
last_name VARCHAR(50) NOT NULL,
email VARCHAR(100) NOT NULL,
age INTEGER);
CREATE TABLE UserPhone (
id INTEGER NOT NULL,
pnumber INTEGER NOT NULL,
ptype VARCHAR(25) NOT NULL,
FOREIGN KEY(id) REFERENCES User(id));
```

Now that I am happy with the schema of the database we can start adding data. I will add three users to the database, two of which don't have a phone number, one with one phone number, and one person who doesn't wish to share their age. To do this I add the users with the following commands:

```
INSERT INTO User (first_name, last_name, email, age)
      values("Braden", "Licastro", "blicastro@live.com", "21");

INSERT INTO User (first_name, last_name, email, age)
      values("Kelly", "Pohland", "myemail@youcantfoolme.com",
      "");

INSERT INTO User (first_name, last_name, email, age)
      values("Justin", "Kirkwood", "kirkwoodj@aol.com", "22");
```

The users have now been added to the user table, now we can add Braden's mobile phone number to the UserPhone table by doing the following:

```
INSERT INTO UserPhone (id, pnumber, ptype) values((SELECT id FROM
      User WHERE first_name = "Braden"), "7244939192", "Mobile");
```

To verify that all of the users and accompanying phone numbers were added correctly, we can look at the entries with the following command:

```
SELECT * FROM User, UserPhone;
1|Braden|Licastro|blicastro@live.com|21|1|7244939192|Mobile
2|Kelly|Pohland|myemail@youcantfoolme.com
3|Justin|Kirkwood|kirkwoodj@aol.com|22
```

In some cases, we may want to remove a specific user from the database. For example the user “Justin Kirkwood” decided to go home for the weekend to avoid Brooks, so we no longer need his entry in the database, and let's assume there is another Justin in the database so we need

to be more specific than just a first name, we can run the following command:

```
DELETE FROM User WHERE first_name="Justin" AND
      last_name="Kirkwood";
```

Now, if you select all users in the database, you will see that user “Justin Kirkwood” has been removed.

```
SELECT * FROM User, UserPhone;
1|Braden|Licastro|blicastro@live.com|21|1|7244939192|Mobile
2|Kelly|Pohland|myemail@youcantfoolme.com
```

And finally, let's say that user “Kelly Pohland” decided it would be beneficial to provide her age. She couldn't remember whether she went by the nickname “Kellogs” or her actual name “Kelly”, we can update the database entry to reflect this change looking for either name using the following code:

```
UPDATE User SET age="19" WHERE first_name LIKE "K%" AND
      last_name="Pohland";
```

This will update any users entry who has a first name that begins with 'K' and has a last name of “Pohland”. To make sure the query worked, we can look at the database entries, and we see that it did indeed work!

```
SELECT * FROM User, UserPhone;
1|Braden|Licastro|blicastro@live.com|21|1|7244939192|Mobile
2|Kelly|Pohland|myemail@youcantfoolme.com|19
```

4. Overall the lab went without a hitch. The most difficult part was creating the writeup and accurately capturing the detailed steps to using SQLite.

   The biggest problem I encountered was using my own laptop. Usually this is not a problem as the software installs fairly easily, but in this case I had another database system installed that conflicted with the SQLite package. In order to install it I had to backup my databases in the other software, upgrade it to a different, compatible version, restore the databases and then I could install SQLite3 package without any trouble. After that everything went smoothly.

   NOTES TO SELF:

   When running complex statements, break into sections and hit enter, entering all on one line sometimes confuses SQLite.

   When canceling a SQLite query that isn't complete DO NOT use ^D or ^Z, it will force quit SQLite or suspend it.

   Next time I work on a different problem, make a new database instead of dropping the table, it will be quicker and you'll be able to reference it at a later time.