SQLite

In this first lab, and several others, we will be using the SQLite relational DBMS (`http://sqlite.org`). SQLite is a light-weight DBMS that will allow us to experiment with SQL and to get a feel for how a database works, without having to go through the more complex interactions involved in using heavier systems like MySQL, PostreSQL, Oracle, etc.

One of SQLite's main advantages is that it stores a database in a single file which can be easily copied and moved around, and accessed directly through a command-line interface or simple APIs. Larger DBMSs also store their data in files, but they are rarely (if ever) manipulated directly by the user/programmer. Instead, they must be accessed through a database server, which is typically not easy to set up. On the other hand, SQLite is simple to install and requires no configuration, so you can start toying and tinkering with it right away, and using an existing database is as easy as making a copy of a database file. More details about SQLite's main features can be found at `http://sqlite.org/different.html` .

SQLite has drawbacks: it does not support high-concurrency applications (e.g., high-traffic websites) or large datasets. Additionally, it does not support all the features of SQL, although this will not be a problem for the simple queries we will be seeing in this lab.

The command `sqlite3` starts a SQLite session:

```
$ sqlite3

SQLite version 3.3.8

Enter ".help" for instructions

sqlite>
```

This starts a SQLite shell where we can type in SQL statements.

Since we started the SQLite shell without accessing an existing database, the only types of statements we could run at this point would be DDL statements (e.g., to create a new table). In this lab you are provided with an example database, planet_express.db; you can download it on BB:

We will first focus on using DML statements to access and manipulate the contents of that database. To start the SQLite shell, and work with the sample database, run the following:

```
$ sqlite3 planet_express.db
```

Besides understanding SQL queries, the SQLite shell also provides several administration commands, which all begin with a dot. Two commands we will find useful are `.tables` and `.schema`. The first one will show a list of all the tables in the database:

```
sqlite> .tables

Client         Has_Clearance  Planet

Employee       Package        Shipment
```

The `.schema` command shows the DDL command that was used to create a table:

```
sqlite> .schema Planet

CREATE TABLE Planet (

  PlanetID INTEGER PRIMARY KEY NOT NULL,

  Name TEXT NOT NULL,

  Coordinates REAL NOT NULL

);
```

Running `.schema` without an argument will dump the entire schema for the database.

Now, letâ€™s run an SQL query that returns the entire contents of the Planet table. Type in the following:

```
sqlite> SELECT * FROM Planet;
```

You should see the following output at the console:

```
1|Omicron Persei 8|89475345.3545

2|Decapod X|65498463216.3466

3|Mars|32435021.65468

4|Omega III|98432121.5464

5|Tarantulon VI|849842198.354654

6|Cannibalon|654321987.21654

7|DogDoo VII|65498721354.688

8|Nintenduu 64|6543219894.1654

9|Amazonia|65432135979.6547
```

By default, SQLite makes no attempt to pretty-print the results of a query (this is good to process the output of a query with another program, but not that good if itâ€™s going to be read by a human). You can make the output easier to read by running the following administration commands:

```
sqlite> .mode column

sqlite> .headers on
```

If you rerun the previous query, you should now see the following:

```
PlanetID    Name              Coordinates

----------  ----------------  -------------

1           Omicron Persei 8  89475345.3545

2           Decapod X         65498463216.3
```

```
3           Mars              32435021.6546

4           Omega III         98432121.5464

5           Tarantulon VI     849842198.354

6           Cannibalon        654321987.216

7           DogDoo VII        65498721354.6

8           Nintenduu 64      6543219894.16

9           Amazonia          65432135979.6
```

Finally, note that you can break a query into as many lines as you want. The end of a query is always delimited by a semicolon. For example:

```
sqlite> SELECT *

    ..> FROM Planet;
```

Sample Queries

Right now we are just learning some basic SQL by rote, so take the following queries with a healthy dose of salt. Their purpose is for you to become familiar with running SQL queries in SQLite and to demonstrate SQLâ€™s simple syntax. For example, the following query shows how you can select some, but not all, of the columns in a table:

```
sqlite> SELECT EmployeeID, Name, Position FROM Employee;
```

You should see the following output:

```
EmployeeID  Name            Position

----------  --------------  ------------

1           Phillip J. Fry  Delivery boy

2           Turanga Leela   Captain
```

```
3           Bender Bending  Robot

4           Hubert J. Farn  CEO

5           John A. Zoidbe  Physician

6           Amy Wong        Intern

7           Hermes Conrad   Bureaucrat

8           Scruffy Scruff  Janitor
```

SQL also allows you to specify conditions on the rows the query will return:

```
sqlite> SELECT EmployeeID, Name, Position FROM Employee

    ..> WHERE Salary >= 10000;
```

The result:

```
EmployeeID  Name           Position

----------  -------------  ----------

2           Turanga Leela  Captain

4           Hubert J. Far  CEO

7           Hermes Conrad  Bureaucrat
```

The `INSERT` command is one of the mechanisms enabling adding new records to the database.

```
sqlite> INSERT INTO Planet(PlanetID, Name, Coordinates)

    ..> VALUES (10, "Jupiter", 1839102.5);
```

Letâ€™s see what happens if we try to violate the primary key integrity of a table. Just try to run the previous INSERT again. This statement should result

in an error (SQL error: PRIMARY KEY must be unique) because there is already a row in the table with a primary key equal to 10.

Exercises

Now try writing some of your own queries. For the first four questions the expected result is shown, so you can verify your work. Use the `.schema`command to find out columns in tables as needed.

1) Select the packages from shipment 3.

```
Shipment    PackageNumber  Contents    Weight      Sender
Recipient

----------  -------------  ----------  ----------  ----------
----------

3           1              Undeclared  15.0        3
4

3           2              Undeclared  3.0         5
1

3           3              Undeclared  7.0         2
3
```

2) Select the packages from shipment 3, with a weight larger than 10.

```
Shipment    PackageNumber  Contents    Weight      Sender
Recipient

----------  -------------  ----------  ----------  ----------
----------

3           1              Undeclared  15.0        3
4
```

3) Select the contents and weight of all the packages.

```
Contents    Weight

----------  ----------
```

```
Undeclared  1.5

Undeclared  10.0

A bucket o  2.0

Undeclared  15.0

Undeclared  3.0

Undeclared  7.0

Undeclared  5.0

Undeclared  27.0

Undeclared  100.0
```

4) Select all the pending shipments. (Testing whether or not a field is empty is done with the boolean expression *`field`* `IS NULL`.)

```
ShipmentID  Date        Manager     Planet

----------  ----------  ----------  ----------

3                       2           3

4                       2           4

5                       7           5
```

5) Add a new client with account number 20 and name "Donbot".