SQL

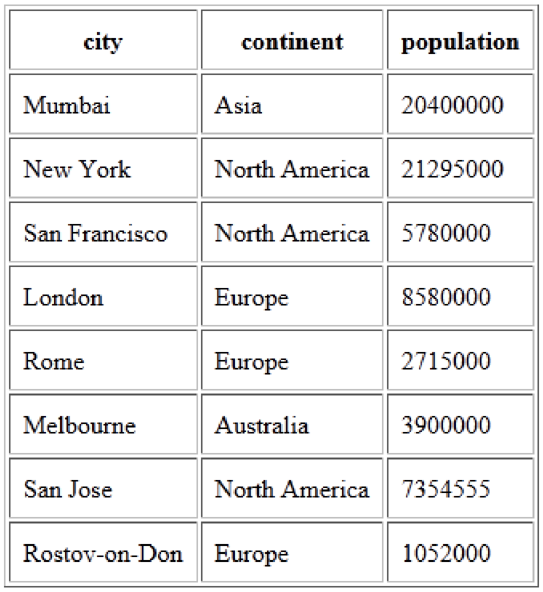
CS106E, Young

A database provides an efficient method for storing and retrieving vast sums of data. Most modern production databases use a special language for storing and retrieving data called SQL. In this handout, I describe using the SQL language with a SQLite database. SQLite is a specific database program which supports SQL. Many databases other than SQLite also use the SQL language. SQLite is widely available and can be used for small to medium-sized websites. Larger websites often use MySQL which uses the same SQL queries listed in this handout, but which requires much greater administrative overhead.

In this handout, I focus on how to interact directly with SQLite by entering commands from what is called the *SQLite shell*. A database administrator might run these commands to interact directly with the database.

Representation of Information

Most modern databases are relational databases. In a relational database, information is stored in tables. For example, we can represent information about major metropolitan areas using the following table:



city	continent	population
Mumbai	Asia	20400000
New York	North America	21295000
San Francisco	North America	5780000
London	Europe	8580000
Rome	Europe	2715000
Melbourne	Australia	3900000
San Jose	North America	7354555
Rostov-on-Don	Europe	1052000

Working with SQLite

In this handout, we'll see how to login into the Stanford UNIX systems in order to create and interact directly with databases. Next week, we'll learn to use the PHP server-side language in conjunction with SQLite to create some interesting websites. Accessing SQLite database from the PHP server-side language will be covered in a different handout.

While ultimately our objective will be to combine SQLite with a website, learning to work with the databases in UNIX will be good practice. Most of the commands we'll experiment with here will also work directly in PHP. Moreover, finding and fixing database errors in a PHP webpage will be much harder than when accessing the database directly. Therefore, having a strong understanding of the basics of creating, editing, and searching through database tables directly will prove a very useful skill for later when we work with webpages

Getting on Stanford's UNIX Systems

You'll need to log in to one of Stanford's UNIX machines. There are several ways to do this, but one simple method is to get a Terminal Emulation program. These programs allow you to emulate a traditional *Computer Terminal*—a computer terminal is a device we used to use to log in to computers back before we had graphical user interfaces and before everyone had their own personal computers.

You can get a Terminal Emulation program from the *Essential Stanford Software* website:

http://ess.stanford.edu/

While you're at the ESS website, you should also pickup a copy of a File Transfer Program, this will prove useful later in the week.

For Windows

If you're using a Windows computer, get the "SecureCRT and SecureFX" package. This will provide you with both terminal emulation and a file transfer program.

Start up your Terminal Emulator. If your Terminal Emulator asks, you want to use the SSH2 protocol and connect through port 22. There are several different Stanford computers you may connect to that will work for this class. We recommend you use "cardinal.stanford.edu". Note that this computer is only to be used for short jobs such as checking email, so don't do any heavy work on it (there are other computers for those purposes).

For Macintosh

If you're using a Macintosh get a copy of Fetch for file transfer. You should already have a Terminal Emulator on your Macintosh. You can get to it by going to Applications > Utilities > Terminal.app.

Run the Terminal.app and enter:

```
ssh yourSUNetID@cardinal.stanford.edu
```

Where *yourSUNetID* is replaced by your 3-8 character long Stanford University Network ID.

Moving to the Correct Directory

Once you've logged into the UNIX system where your database is, you'll want to move to the directory where you want to place your database file (or to the directory where you have a pre-existing SQLite database file). You can change directories in UNIX using the cd command. For example, if I want to move into the cgi-bin directory I would enter:

```
> cd cgi-bin
```

To change to a directory named WWW I would type:

```
> cd WWW
```

If you want to move to a higher directory, you can type:

```
> cd ..
```

If you end up getting lost you can return to your top level directory by entering:

```
> cd ~/
```

You can see all the files and directories available by entering:

```
> ls
```

If you want to create a new directory you use the mkdir (Make Directory) command. Here I am creating a new directory called test:

```
> mkdir test
```

Starting Up SQLite3

Once you are in the directory you want to use, start SQLite by typing:

```
> sqlite3 customers.db
```

The '3' here is the version number of SQLite that we are using. Make sure you include it, older versions of SQLite are not compatible with SQLite version 3 files. customers.db is the name of the database file we want to work with.

You would use the same command to create a new database file. For example, to create a brand-new cities database file we would enter:

```
> sqlite3 cities.db
```

Once you run the sqlite3 command you will be in the sqlite3 command shell. The UNIX directory commands from the previous section will not be available until you quit sqlite3.

Enter the following two commands to set SQLite's formatting:

```
sqlite> .mode column
sqlite> .headers on
```

These two commands tell SQLite3 to display data using carefully aligned columns and to list the column names as headers above each column. When using the SQLite shell you'll see two types of commands, one type will be preceded by a period as shown here. These commands are specific to SQLite and can only be entered at the SQLite shell. The other commands, which we'll see in a moment do not have periods preceding them, these commands are general SQL commands and can be used in PHP as well as in other databases such as MySQL which support the SQL language. We will also see that SQL commands end with a semicolon ';' whereas SQLite-specific commands do not.

Two more commands before we start creating tables in our database. You can type .help to get a listing of all the SQLite commands.

```
sqlite> .help
```

Finally when you're all done, you can exit from the SQLite shell by using .quit

```
sqlite> .quit
```

Creating a Table

When you first create a database file using SQLite it is empty. We will now create a table and add it to our database. To define a table, we need to give the table a name, and determine what columns will be in the table. Here is an example:[1]

```
sqlite> CREATE TABLE cities (
   ...> city TEXT,
   ...> continent TEXT,
   ...> population INTEGER
   ...> );
```

We've created a table with three columns, a city name column which is text, a continent column which is also text, and a population column which stores integers. In addition to the TEXT and INTEGER types we've used here, there is also a REAL type which can be used to store floating point numbers such as 3.14159.

Our next step is to populate our table. We do this using INSERT commands:

```
sqlite> INSERT INTO cities VALUES(
   ...> 'Mumbai','Asia',20400000);
sqlite> INSERT INTO cities VALUES(
   ...> 'New York','North America',21295000);
sqlite> INSERT INTO cities VALUES(
   ...> 'San Francisco','North America',5780000);
```

As you can see to INSERT we specify the name of the table we are inserting into, followed byVALUES, then a list of values, one for each of the columns in our table.

If you need to insert a lot of items you can do it with a single INSERT while separating entries with commas like this:

```
sqlite> INSERT INTO cities VALUES
   ...> ('London','Europe',8580000),
   ...> ('Rome','Europe',2715000),
   ...> ('Melbourne','Australia',3900000),
   ...> ('San Jose','North America',7354555),
   ...> ('Rostov-on-Don','Europe',1052000);
```

Using the INSERT command we have now entered into our database the table described on the first page of this handout.

Note that while we've used the same name for our table as for our database file, this is not a requirement. Furthermore, as we'll see later in the handout, a database may contain many tables.

[1] The "sqlite>" is the standard SQLite command prompt. The "…>" is the SQLite prompt used to show line continuation. In this case, if we enter "CREATE TABLE metropolises (" on a line, SQLite knows that it is not a complete statement, and it uses the "…>" prompt characters to indicate that it is waiting for additional input before executing the statement.

Displaying and Searching with a Single Table

We can display the contents of a table using the SELECT statement. Using a variety of "clauses" added to a SELECT statement, we control which rows and columns are displayed and the order in which they are displayed.

Displaying Table Data (Restricting by Columns)

The simplest form of SELECT lists all the rows in a table, allowing the user to determine which columns to display. For example here is simple SELECT statement:

```
SELECT population FROM cities;
```

This tells SQLite that we want to display items from the cities database. We are only interested in seeing the population column and we do not have any restrictions on which rows to display. Typing this in gives us the following result:

```
sqlite> SELECT population FROM cities;

population
---------------
20400000
21295000
5780000
8580000
2715000
3900000
7354555
1052000
```

We can instruct SQLite to display as many specific columns as we want by listing the columns, separated by commas. Here for example we list two columns:

```
sqlite> SELECT city,continent FROM cities;

city             continent
---------------  ---------------
Mumbai           Asia
New York         North America
San Francisco    North America
London           Europe
Rome             Europe
Melbourne        Australia
San Jose         North America
Rostov-on-Don    Europe
```

Displaying All Columns

If we want to display all columns we can use * in place of the column names. This query, for example, lists all columns and all rows in the cities database:

```
sqlite> SELECT * FROM cities;

city             continent        population
---------------  ---------------  ---------------
Mumbai           Asia             20400000
New York         North America    21295000
San Francisco    North America    5780000
London           Europe           8580000
Rome             Europe           2715000
Melbourne        Australia        3900000
San Jose         North America    7354555
Rostov-on-Don    Europe           1052000
```

Controlling Column Width
You may find that your data has too many characters to fit in the space SQLite has allotted for it. In this case, SQLite will cut off the display of the extra characters like this:

```
city        continent   population
----------  ----------  ----------
Mumbai      Asia        20400000
New York    North Amer  21295000
San Franci  North Amer  5780000
London      Europe      8580000
Rome        Europe      2715000
Melbourne   Australia   3900000
San Jose    North Amer  7354555
Rostov-on-  Europe      1052000
```

You can instruct SQLite to change the column widths using the .width command. Use this to specify the width of each column. Here we've instructed SQLite to display city and continent as 15-character wide columns and population as a 10-character wide column.

```
sqlite> .width 15 15 10
```

Displaying Table Data (Restricting Rows)
We can tell SQLite that we are only interested in some of the rows by adding in a WHERE clause. For example:

```
sqlite> SELECT * FROM cities WHERE continent = 'Europe';

city             continent        population
---------------  ---------------  ---------------
London           Europe           8580000
Rome             Europe           2715000
Rostov-on-Don    Europe           1052000
```

Using Comparison Operators
The WHERE clause supports a variety of different comparison operators. It uses = for equals (note that's only a single equal sign not the double one you are used to), != for not equals (SQL also accepts <> for not equals), >, >=, <, and <=. As with PHP you may use >, >=, <, and <= to alphabetize strings in addition to using them with numbers. Here is an example using a comparison operator:

```
sqlite> SELECT * FROM cities WHERE population > 20000000;

city               continent          population
---------------    ---------------    ---------------
Mumbai             Asia               20400000
New York           North America      21295000
```

Searching for Partial Matches

The WHERE clause allows us to search for partial matches using some special characters. To use these, list the column whose values you are trying to match with followed by the keyword LIKE and then use the following rules:

- A % indicates a wild card character that can be replaced by any sequence of characters. For example

  ```
  sqlite> SELECT * FROM cities WHERE city LIKE 'San%';
  ```

  matches all metropolises which start with the letters "San" and generates the result:

  ```
     city               continent          population
     ---------------    ---------------    ---------------
     San Francisco      North America      5780000
     San Jose           North America      7354555
  ```

- An underscore '_' represents a single character which will successfully match against any character.

  ```
  sqlite> SELECT * FROM cities WHERE city LIKE '_o%';
  ```

  matches against metropolises which start any given letter, have a second letter which is an 'o' and end with any sequence of letters. This generates the result:

  ```
     city               continent          population
     ---------------    ---------------    ---------------
     London             Europe             8580000
     Rome               Europe             2715000
     Rostov-on-Don      Europe             1052000
  ```

See here for a list of other special characters used in SQL matching:

   http://dev.mysql.com/doc/refman/5.5/en/regexp.html

Combining with Boolean Operators

SQL supports AND, OR, XOR, and NOT operators. Here we find all metropolises which start with "San" or "New":

```
sqlite> SELECT * FROM cities
   ...> WHERE city LIKE 'San%' OR city LIKE 'New%';

city             continent        population
---------------  ---------------  ---------------
New York         North America    21295000
San Francisco    North America    5780000
San Jose         North America    7354555
```

You can control order of evaluation of the Boolean operators using parentheses.

Controlling Output Order

You can control the order in which rows are listed using the ORDER BY clause. Here we ask SQLite to list all our metropolises using the population to determine order:

```
sqlite> SELECT * FROM cities ORDER BY population;

city             continent        population
---------------  ---------------  ---------------
Rostov-on-Don    Europe           1052000
Rome             Europe           2715000
Melbourne        Australia        3900000
San Francisco    North America    5780000
San Jose         North America    7354555
London           Europe           8580000
Mumbai           Asia             20400000
New York         North America    21295000
```

Reverse order by adding a DESC (short for descending) after the name of the column you are using to order. ORDER BY can be combined with the other clauses we've already seen. In this example we list metropolises which have populations over 50,000,000 in order from largest to smallest:

```
sqlite> SELECT * FROM cities
   ...> WHERE population > 5000000 ORDER BY population DESC;

city             continent        population
---------------  ---------------  ---------------
New York         North America    21295000
Mumbai           Asia             20400000
London           Europe           8580000
San Jose         North America    7354555
San Francisco    North America    5780000
```

Combining Tables (Joins)

Most databases will contain information in multiple tables. SQL provides a variety of methods for combining information across tables. Suppose in addition to our cities table, we have a second table containing information about universities. The table looks like this:

```
university                      city
------------------------------  ---------------
Stanford University             San Francisco
Columbia University             New York
Juilliard School                New York
Fordham University              New York
Harvard                         Boston
University of the Arts London   London
London School of Economics      London
University of the Arts          London
```

Notice that while this table includes the city in which the universities are located, it does not duplicate the continent and population information from our previous table. Also notice that this table includes one city, Boston, which is not in the cities table. Similarly the cities table includes quite a number of cities without universities listed. These facts will become significant in a moment.

We can use a JOIN expression to work with multiple tables. There are several versions of JOIN.

INNER JOIN

The INNER JOIN expression allows us to combine two tables listing rows where a specific column matches. Here is an example:

```
sqlite> SELECT * FROM cities INNER JOIN universities USING (city);

city             continent        population  university
---------------  ---------------  ----------  ------------------------------
New York         North America    21295000    Columbia University
New York         North America    21295000    Fordham University
New York         North America    21295000    Juilliard School
San Francisco    North America    5780000     Stanford University
London           Europe           8580000     London School of Economics
London           Europe           8580000     University of the Arts
London           Europe           8580000     University of the Arts London
```

We instructed SQLite to combine information from the cities table with that from the universities table, by finding rows where the city value matched.

Notice that Harvard is not listed, as there was no match for the city “Boston” in the cities table. Similarly notice that none of the cities without universities are listed.

While our tables aren’t complex enough to warrant it, if you have tables where you only want items listed if you have more than one column matching you can add the additional columns inside the parenthesis of the USING clause.

LEFT JOIN and RIGHT JOIN

While INNER JOIN only lists rows with column matches, the LEFT JOIN operation tells the database to list every row from the left table, even if the row doesn’t match with a row in the right table. We still provide it with one or more columns, so that it knows what to use when matching the two tables. Here is a LEFT JOIN. In this case the table metropolises is the left table and universities is the right table (metropolises is listed to the left of the JOIN keyword and universities appears to the right of the JOIN keyword). This LEFT JOIN tells SQLite to list all entries in the left table, but only those in the right table that match rows in the left table.

```
sqlite> SELECT * FROM cities LEFT JOIN universities USING (city);

city             continent        population  university
---------------  ---------------  ----------  ------------------------------
Mumbai           Asia             20400000
New York         North America    21295000    Columbia University
New York         North America    21295000    Fordham University
New York         North America    21295000    Juilliard School
San Francisco    North America    5780000     Stanford University
London           Europe           8580000     London School of Economics
London           Europe           8580000     University of the Arts
London           Europe           8580000     University of the Arts London
Rome             Europe           2715000
Melbourne        Australia        3900000
San Jose         North America    7354555
Rostov-on-Don    Europe           1052000
```

As you can see, every metropolis is listed, whether or not it had a corresponding university. You'll notice that Harvard is still not listed.

RIGHT JOIN would do the opposite, listing everything in the universities table, whether or not there was a corresponding entry in the metropolises table.

Modifying a Table

There are a variety of methods available for modifying a database.

INSERT

We've already seen use of INSERT. It's how we built our databases. Here is the basic INSERT:

```
sqlite> INSERT INTO cities VALUES('Mumbai','Asia',20400000);
```

As you can see we simply list the name of the table followed by all the values to place in the table.

UPDATE

UPDATE can be used to change rows in a table. UPDATE sets values in all rows which match a particular criteria:

```
UPDATE cities SET continent = 'EU' WHERE continent = 'Europe';
```

This tells the system to update the metropolises table. Each entry which matches the WHERE criteria should have its continent column set to "EU".

You can update individual rows by using sufficiently tight matching criteria. For example the following sets a single data cell "Rome" to "Roma":

```
UPDATE cities SET city = 'Roma' WHERE city = 'Rome';
```

DELETE

Delete does exactly what you would expect – it removes rows from your table. You can use the same WHERE clause we used with SELECT in order to determine which rows to remove. If we enter:

```
DELETE FROM cities WHERE continent = 'North America';
```

Our revised table looks like this, with all metropolises in North America removed.

```
city             continent        population
---------------  ---------------  ----------
Mumbai           Asia             20400000
London           Europe           8580000
Rome             Europe           2715000
Melbourne        Australia        3900000
Rostov-on-Don    Europe           1052000
```

Working with Database Files and Text Files

We will occasionally give you files containing SQLite commands which you can copy and paste into the SQLite shell. SQLite doesn't actually have a specified file extension. So we will use the following convention. Actual SQLite databases which we create by running the sqlite3 command we will give the extension *.db. Text files containing SQL commands we will give the extension *.txt.

It is possible to load a text file and execute all the SQL commands in it. If we give you a text file "examples.txt" containing SQL commands, you can either copy and paste each line one a time into the SQLite shell or you can use the ".run" command from within the shell:

```
sqlite> .run examples.txt
```

When transferring *.db and *.txt files your file transfer program may ask if the file type being transferred is a text file or a binary file.[2] Use text for HTML, CSS, PHP, and of course Text files. Use binary for the *.db files.

[2] You may recall from our very first lecture when we discussed character encodings that I mentioned that Unix, Macs, and Windows all have different methods for indicating the end of line.