MySQL

This handout briefly describes how to work with Stanford's MySQL database server directly. Handout #21 will describe how to access the database from Java.

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:

metropolis	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 MySQL

We will be working with Stanford's MySQL database. At this point you should have received an e-mail containing the following pieces of information:

- Your MySQL Username
- Your MySQL Initial Password
- The name of the Stanford MySQL Database Server
- Your MySQL Database Name

Keep this information handy, you'll need it as you work with MySQL.

Logging In

To use MySQL login to a Stanford UNIX computer and enter the following at the command prompt:

```
mysql –h mysql-user.stanford.edu -u userID –p
```

where you replace *userID* with your MySQL user ID. The system will prompt you for a password. Note that the User ID and Password for MySQL are not the same as your regular Stanford ID and password (and you should keep it that way for security reasons).

Once you've logged in, notify MySQL which database you want to use. With Stanford's MySQL setup, each user has been assigned a specific database that they can use. This is the database name you received via e-mail along with your username and password. We tell MySQL which database to use like this:

```
mysql> USE database_name;
```

where *database_name* is replaced by the database name you have been assigned.

Creating a Table

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]

```
mysql> CREATE TABLE metropolises (
    -> metropolis CHAR(64),
    -> continent CHAR(64),
    -> population BIGINT
    ->);
```

We've created a table with three columns, a metropolis column composed of 64-character long strings, a continent column with 64-character long strings, and a population column which is a 64-bit integer.

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

```
mysql> INSERT INTO metropolises VALUES(
    -> "Mumbai","Asia",20400000);
mysql> INSERT INTO metropolises VALUES(
    -> "New York","North America",21295000);
mysql> INSERT INTO metropolises VALUES(
    -> "San Francisco","North America",5780000);
```

[1] The "mysql>" is the standard MySQL command prompt. The "->" is the MySQL prompt used to show line continuation. In this case, if we enter "CREATE TABLE metropolises (" on a line, MySQL 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.

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

```
mysql> INSERT INTO metropolises 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 the table described on the first page into our database.

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 metropolises;
```

This tells MySQL that we want to display items from the metropolises 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:

```
mysql> SELECT population FROM metropolises;
+------------+
| population |
+------------+
| 20400000   |
| 21295000   |
| 5780000    |
| 8580000    |
| 2715000    |
| 3900000    |
| 7354555    |
| 1052000    |
```

```
+------------+
8 rows in set (0.02 sec)
```

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

```
mysql> SELECT metropolis,continent FROM metropolises;
+---------------+---------------+
| metropolis    | 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        |
+---------------+---------------+
8 rows in set (0.00 sec)
```

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 metropolises database:

```
mysql> SELECT * FROM metropolises;
+---------------+---------------+------------+
| metropolis    | 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    |
+---------------+---------------+------------+
8 rows in set (0.00 sec)
```

Displaying Table Data (Restricting Rows)

We can tell MySQL that we are only interested in some of the rows by adding in a WHERE clause. For example:

```
mysql> SELECT * FROM metropolises WHERE continent = "Europe";
+---------------+-----------+------------+
| metropolis    | continent | population |
+---------------+-----------+------------+
| London        | Europe    | 8580000    |
| Rome          | Europe    | 2715000    |
| Rostov-on-Don | Europe    | 1052000    |
+---------------+-----------+------------+
3 rows in set (0.00 sec)
```

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 Java you may use >, >=, <, and <= to alphabetize strings in addition to using them with numbers. Here is an example using a comparison operator:

```
mysql> SELECT * FROM metropolises WHERE population > 20000000;
+------------+---------------+------------+
| metropolis | continent     | population |
+------------+---------------+------------+
| Mumbai     | Asia          | 20400000   |
| New York   | North America | 21295000   |
+------------+---------------+------------+
2 rows in set (0.01 sec)
```

Using Regular Expressions

The WHERE clause also supports regular expressions. 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. For example

```
mysql> SELECT * FROM metropolises
    -> WHERE metropolis LIKE "San%"
```

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

```
+---------------+---------------+------------+
| metropolis    | continent     | population |
+---------------+---------------+------------+
| San Francisco | North America | 5780000    |
| San Jose      | North America | 7354555    |
+---------------+---------------+------------+
2 rows in set (0.04 sec)
```

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

  ```
  mysql> SELECT * FROM metropolises
      -> WHERE metropolis 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:

```
+---------------+-----------+------------+
| metropolis    | continent | population |
+---------------+-----------+------------+
| London        | Europe    | 8580000    |
| Rome          | Europe    | 2715000    |
| Rostov-on-Don | Europe    | 1052000    |
+---------------+-----------+------------+
3 rows in set (0.00 sec)
```

Combining with Boolean Operators

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

```
mysql> SELECT * FROM metropolises
    -> WHERE metropolis LIKE "San%" OR metropolis LIKE "New%";
+---------------+---------------+------------+
| metropolis    | continent     | population |
+---------------+---------------+------------+
| New York      | North America | 21295000   |
| San Francisco | North America | 5780000    |
| San Jose      | North America | 7354555    |
+---------------+---------------+------------+
3 rows in set (0.00 sec)
```

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 MySQL to list all our metropolises using the population to determine order:

```
mysql> SELECT * FROM metropolises ORDER BY population;
+---------------+---------------+------------+
| metropolis    | 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   |
+---------------+---------------+------------+
8 rows in set (0.00 sec)
```

Reverse order by adding a DESC 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:

```
mysql> SELECT * FROM metropolises
    -> WHERE population > 5000000 ORDER BY population DESC;
+---------------+---------------+------------+
| metropolis    | continent     | population |
+---------------+---------------+------------+
| New York      | North America | 21295000   |
| Mumbai        | Asia          | 20400000   |
| London        | Europe        | 8580000    |
| San Jose      | North America | 7354555    |
| San Francisco | North America | 5780000    |
+---------------+---------------+------------+
5 rows in set (0.00 sec)
```

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 metropolises table, we have a table containing information about universities. The table looks like this:

```
+------------------------------+---------------+
| university                   | metropolis    |
+------------------------------+---------------+
| 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 metropolis where universities are located, it does not duplicate the continent and population information from our previous table. Also notice that this table includes a metropolis, Boston, which is not in the metropolises table. Similarly the metropolises table includes quite a number of metropolises 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:

```
mysql> SELECT * FROM metropolises INNER JOIN universities USING (metropolis);
+---------------+---------------+------------+-------------------------------+
| metropolis    | continent     | population | university                    |
+---------------+---------------+------------+-------------------------------+
| San Francisco | North America | 5780000    | Stanford University           |
| New York      | North America | 21295000   | Columbia University           |
| New York      | North America | 21295000   | Juilliard School              |
| New York      | North America | 21295000   | Fordham University            |
| London        | Europe        | 8580000    | University of the Arts London |
| London        | Europe        | 8580000    | London School of Economics    |
| London        | Europe        | 8580000    | University of the Arts        |
+---------------+---------------+------------+-------------------------------+
7 rows in set (0.03 sec)
```

We instructed MySQL to combine information from the metropolises table with that from the universities table, by finding rows where the metropolis value matched.

Notice that Harvard is not listed, as there was no match for the metropolis “Boston” in the metropolises table. Similarly notice that none of the metropolises 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 MySQL to list all entries in the left table, but only those in the right table that match rows in the left table.

```
mysql> SELECT * FROM metropolises LEFT JOIN universities USING (metropolis);
+---------------+---------------+------------+--------------------------------+
| metropolis    | continent     | population | university                     |
+---------------+---------------+------------+--------------------------------+
| Mumbai        | Asia          | 20400000   |                                |
| New York      | North America | 21295000   | Columbia University            |
| New York      | North America | 21295000   | Juilliard School               |
| New York      | North America | 21295000   | Fordham University             |
| San Francisco | North America | 5780000    | Stanford University            |
| London        | Europe        | 8580000    | University of the Arts London  |
| London        | Europe        | 8580000    | London School of Economics     |
| London        | Europe        | 8580000    | University of the Arts         |
| Rome          | Europe        | 2715000    |                                |
| Melbourne     | Australia     | 3900000    |                                |
| San Jose      | North America | 7354555    |                                |
| Rostov-on-Don | Europe        | 1052000    |                                |
+---------------+---------------+------------+--------------------------------+
12 rows in set (0.00 sec)
```

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:

```
mysql> INSERT INTO metropolises 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.

There is a fancier form of INSERT, but in order to use it, we'll need to enhance our understanding of table definition. Previously we defined our table like this:

```
CREATE TABLE metropolises (
    metropolis CHAR(64),
    continent CHAR(64),
    population BIGINT
);
```

If we want, we can provide a default value for one or more of the columns. Here is a similar table, where we've told SQL that if continent is not provided, it should default to "North America".

```
CREATE TABLE metro2 (
    metropolis CHAR(64),
    continent CHAR(64) DEFAULT "North America",
    population BIGINT
);
```

We can now INSERT into the metro2 table, without providing all three pieces of information. Here's the format:

```
INSERT INTO metro2(metropolis, population) VALUES ("Seattle",3344813);
```

As you can see we add a pair of parentheses after the table name notifying MySQL which column values we will be providing. We then list only those items in the VALUES.

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 metropolises WHERE continent = "North America";
```

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

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

UPDATE

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

```
UPDATE metropolises 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 metropolises SET metropolis = "Roma"
WHERE metropolis = "Rome";
```

Working with SQL Files

Typically our data will be entered into the database, and the database will maintain persistence of the data – persistence is after all, one of the key purposes of a database. For testing purposes, however, we will find it particularly useful to quickly and easily reset the database contents – clearing any tables we've modified and then execute all the SQL commands needed to get the data back into its original state. We can do this by storing MySQL commands in a file. Here is a sample SQL file:

```
USE c_cs108_student;

DROP TABLE IF EXISTS metropolises;
 -- remove table if it already exists and start from scratch

CREATE TABLE metropolises (
    metropolis CHAR(64),
    continent CHAR(64),
    population BIGINT
);

INSERT INTO metropolises VALUES
   ("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);
```

Where *c_cs108_student* is the name of the database we have been assigned on the Stanford server. You'll notice this is essentially exactly the commands that we typed in in order to create our metropolises table, with the addition of the line:

```
DROP TABLE IF EXISTS metropolises;
```

As noted in the comment in the file, the purpose of this line is to remove any pre-existing metropolises table. The IF EXISTS is there because otherwise we will get an error message if we try to DROP the table and it does not exist.

To load a SQL file like this you simply use the SOURCE command, as follows:

```
mysql> SOURCE metropolises.sql;
```

Helpful Hints

- Don't forget you need to call USE in order to select the particular database you want to work with.
- If you forget the name of your database you can query for database names using SHOW DATABASES;
- If you forget the name of a table, you can query for the names of all tables in the active database using SHOW TABLES;
- SQL commands are case insensitive. Database and table names may or may not be case-sensitive depending on which platform you are using (e.g., Windows vs. Unix). Column names are case insensitive. Matching is generally case insensitive.
- Exit MySQL by using QUIT;