

Session Six -- March 28, 2005

Note: See <http://dev.mysql.com/doc/mysql/en/> and <http://php.net/manual/en/> for complete technical documentation

Outline

- connecting with MySQL with the command line client
- MySQL data types
- creating tables and databases
- SQL syntax: inserting, deleting, updating, and retrieving data
- the MySQL user/privilege system: how to create a user with GRANT
- connecting and querying via the PHP mysqli API

Database is one of those overloaded terms like "server" Just as *server* sometimes means a machine like `www123.yourisp.com` and sometime it means a program like Apache httpd and sometimes it means both -- so it is with *database*, as it used commonly used in reality (as opposed to what the dictionary says): sometimes it means a database server program like MySQL, sometimes it means a collection of data tables.

One definition of database that I found: an organized body of related information. For an interesting read see <http://www.google.com/search?hl=en&q=define%3Adatabase>

Pronunciation guide

SQL: I say *sequel*.

MySQL: Officially it is *my ess cue el* (*my* as in *my*, *ess* as in *escrow*, *cue* as in *pool cue*, *el* as in *elbow*) but a lot people say *my sequel*. Take your pick, then expect to be corrected whichever way you pronounce it.

What is SQL?

Originally standing for Structured Query Language, now it pretty much just means SQL. It was developed by IBM in the mid-1970s as a way to get information into and out of relational database management systems in mainframes. Although SQL is both an ANSI and an ISO standard, many database products support SQL with proprietary extensions to the standard language. In other words all your major databases support a big subset of standard SQL, and they all have their own idiolects that are not necessarily compatible with one another.

SQL was originally intended to make it possible for people with only a modicum of technical expertise to query databases. It's a high-level language which at its simplest reads a lot like English:

```
SELECT lastname, firstname, email FROM people WHERE zipcode = '07302'
```

Without knowing much about SQL, but with the knowledge that there is a database table called 'people' with columns called *lastname*, *firstname*, *email*, and *zipcode*, you might guess from the above that executing this SQL statement against a database server would make it fetch from this table all the lastnames, firstnames and email addresses from the rows whose the zipcode field was 07302.

What is MySQL?

PHP supports numerous RDBMSes – *relational database management systems* – including Oracle, PostgreSQL, MS SQL Server, Sybase, and more. MySQL is an extremely popular, fast, stable, powerful open source RDBMS that works well with PHP. For details please see <http://mysql.com/>

We should point out that the spelling **MySQL** denotes the whole product; lowercase **mysql** is the name of the client program, but occasionally people use it as a synonym for MySQL; and the name of the server executable itself is **mysqld** – `mysqld*.exe` on Windows. MySQL installations also use a database called `mysql` to keep track of administrative data and metadata.

MySQL is an immense subject: there are 1200 page books about it, and a certification program for it. We can't get as deeply into it as we might like but we will learn enough more than enough to get started.

Step 1: Install MySQL 4.1.x on your development machine if you haven't already. (If you got the Week 1 assignment done properly, then this is already done).

Step 2: connect with it from the command line with the `mysql` client. (again, `mysqld` is the name of the server program; `mysql` is the client program). You launch an instance of the client by entering its name at a terminal window command line. The directory containing the MySQL command line client executable – called simply **mysql** on Unix-like platforms, **mysql.exe** on Windows -- needs to be in your PATH environment variable, or else you have to invoke it by its full path so the shell will know where to find it -- same as with any other program.

There are numerous command line options, but the most essential options and syntax are

```
mysql -u<username> -h<hostname> -p[password] [database_name]
```

If you leave out the `-u` option MySQL assumes the current shell user. If you leave out the host `mysql` assumes `localhost` – i.e., the machine you're logged onto. If you give it the `-p` option but no password, `mysql` prompts you for it.

If the MySQL installation is properly secured, you will need both a username and password, so use these to log in. If you find that you can log in just by typing `mysql`, your installation has not been secured (see below).

If you supply the `database_name` argument, MySQL will try to connect you to that specific database. Otherwise, after you connect, you will need to tell MySQL which database you want to talk to by saying `use database_name`.

Here is an failed login on my Windows box

```
MINTZD# mysql
ERROR 1045: Access denied for user: 'ODBC@localhost' (Using password: NO)
```

Once again, with a valid username and password:

```
MINTZD# mysql -uroot -p
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 15 to server version: 4.0.20a-debug-log

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

Now I am going to use a particular database (the one I am using for development for the US District Court interpreters office, called `interp_dev`)

```
mysql> use interp_dev
Database changed
mysql>
```

I am can now execute SQL queries against that database by typing them at the command line, followed by a semicolon, and pressing Enter. The results are displayed on the terminal.

Securing a new MySQL installation

If you were able to log in without a username and password, you probably should secure your installation. At a minimum, that means running this command (with your `mysqld` server running):

```
/usr/local/mysql/bin/mysqladmin -u root password new_password_here
```

Followed by

```
/usr/local/mysql/bin/mysqladmin -u -p root reload
```

This has the effect of setting a password for the root user, and getting `mysql` to reload its *grant tables* to make the change take effect. We will deal with the MySQL privilege system in a bit more detail later.

Creating Databases

Before you can do anything with a database, obviously it has to exist. The MySQL command to create a database is easy: `create database <database_name>`. Suppose we decide to establish a University of PHP and we know we are going to have students, faculty, and courses. First create the database

```
mysql>CREATE DATABASE php_university;
```

```
Query OK, 1 row affected (0.01 sec)
mysql>
```

Creating Tables

The next thing you will need to do is create some tables to hold our data. You do this with the "create table <table_name> ..." statement. This command requires a table *name* followed by the table *definition*: column names, data types, etc. That leads us naturally into a discussion of some of the MySQL datatypes.

MySQL Data Types

MySQL is not loosely typed like PHP, so you can't just say "create table student (lastname, firstname);" or you will get something like this (which btw will probably become all too familiar):

```
mysql> create table student (lastname, firstname);
ERROR 1064: You have an error in your SQL syntax. Check the manual that
corresponds to your MySQL server version for the right syntax to use near '
firstname)'
at line 1
mysql>
```

You have to explicitly tell MySQL what kind of data you want to store in which column at table creation time – though you can also change your mind later with the ALTER TABLE command.

Please see http://dev.mysql.com/doc/mysql/en/Column_types.html etc for the complete story as told by the ultimate authoritative source, or consult a reputable reference book. Much of the following sections is borrowed verbatim from the MySQL documentation site.

In the most general terms, there are three types: **numeric**, **string**, and **date/time**.

In the following synopsis, M indicates the maximum display width. The maximum legal display width is 255. D applies to floating-point and fixed-point types and indicates the number of digits following the decimal point. The maximum possible value is 30, but should be no greater than M-2. Square brackets ([and]) indicate parts of type specifiers that are optional.

Numeric types

TINYINT[(M)] [UNSIGNED] [ZEROFILL]

A very small integer. The signed range is -128 to 127. The unsigned range is 0 to 255.

BIT

BOOL

BOOLEAN

These are synonyms for TINYINT(1). The BOOLEAN synonym was added in MySQL 4.1.0. A value of zero is considered false. Non-zero values are considered true. In the future, full boolean type handling will be introduced in accordance with standard SQL.

SMALLINT[(M)] [UNSIGNED] [ZEROFILL]

A small integer. The signed range is -32768 to 32767. The unsigned range is 0 to 65535.

MEDIUMINT[(M)] [UNSIGNED] [ZEROFILL]

A medium-size integer. The signed range is -8388608 to 8388607. The unsigned range is 0 to 16777215.

INT[(M)] [UNSIGNED] [ZEROFILL]

A normal-size integer. The signed range is -2147483648 to 2147483647. The unsigned range is 0 to 4294967295.

INTEGER[(M)] [UNSIGNED] [ZEROFILL]

A synonym for INT.

DECIMAL[(M[,D])] [UNSIGNED] [ZEROFILL]

An unpacked fixed-point number. Behaves like a CHAR column; "unpacked" means the number is stored as a string, using one character for each digit of the value. M is the total number of digits and D is the number of decimals. The decimal point and (for negative numbers) the '-' sign are not counted in M, although space for them is reserved. If D is 0, values have no decimal point or fractional part. The maximum range of DECIMAL values is the same as for DOUBLE, but the actual range for a given DECIMAL column may be constrained by the choice of M and D. If UNSIGNED is specified, negative values are disallowed. If D is omitted, the default is 0. If M is omitted, the default is 10. Prior to MySQL 3.23, the M argument must be large enough to include the space needed for the sign and the decimal point.

(There are more numeric types than this. See http://dev.mysql.com/doc/mysql/en/Numeric_type_overview.html)

Date/time types

DATE

A date. The supported range is '1000-01-01' to '9999-12-31'. MySQL displays DATE values in 'YYYY-MM-DD' format, but allows you to assign values to DATE columns using either strings or numbers.

DATETIME

A date and time combination. The supported range is '1000-01-01 00:00:00' to '9999-12-31 23:59:59'. MySQL displays DATETIME values in 'YYYY-MM-DD HH:MM:SS' format, but allows you to assign values to DATETIME columns using either strings or numbers.

TIMESTAMP[(M)]

A timestamp. The range is '1970-01-01 00:00:00' to partway through the year 2037. A TIMESTAMP column is useful for recording the date and time of an INSERT or UPDATE operation. The first TIMESTAMP column in a table is automatically set to the date and time of the most recent operation if you don't assign it a value yourself. You can also set any

TIMESTAMP column to the current date and time by assigning it a NULL value. From MySQL 4.1 on, TIMESTAMP is returned as a string with the format 'YYYY-MM-DD HH:MM:SS'. If you want to obtain the value as a number, you should add +0 to the timestamp column. Different timestamp display widths are not supported. In MySQL 4.0 and earlier, TIMESTAMP values are displayed in YYYYMMDDHHMMSS, YMMDDHHMMSS, YYYYMMDD, or YMMDD format, depending on whether M is 14 (or missing), 12, 8, or 6, but allows you to assign values to TIMESTAMP columns using either strings or numbers. The M argument affects only how a TIMESTAMP column is displayed, not storage. Its values always are stored using four bytes each. From MySQL 4.0.12, the --new option can be used to make the server behave as in MySQL 4.1. Note that TIMESTAMP(M) columns where M is 8 or 14 are reported to be numbers, whereas other TIMESTAMP(M) columns are reported to be strings. This is just to ensure that you can reliably dump and restore the table with these types.

TIME

A time. The range is '-838:59:59' to '838:59:59'. MySQL displays TIME values in 'HH:MM:SS' format, but allows you to assign values to TIME columns using either strings or numbers.

YEAR[(2|4)]

A year in two-digit or four-digit format. The default is four-digit format. In four-digit format, the allowable values are 1901 to 2155, and 0000. In two-digit format, the allowable values are 70 to 69, representing years from 1970 to 2069. MySQL displays YEAR values in YYYY format, but allows you to assign values to YEAR columns using either strings or numbers. The YEAR type is unavailable prior to MySQL 3.22.

String types

[NATIONAL] CHAR(M) [BINARY | ASCII | UNICODE]

A fixed-length string that is always right-padded with spaces to the specified length when stored. M represents the column length. The range of M is 0 to 255 characters (1 to 255 prior to MySQL 3.23). Note: Trailing spaces are removed when CHAR values are retrieved. From MySQL 4.1.0, a CHAR column with a length specification greater than 255 is converted to the smallest TEXT type that can hold values of the given length. For example, CHAR(500) is converted to TEXT, and CHAR(200000) is converted to MEDIUMTEXT. This is a compatibility feature. However, this conversion causes the column to become a variable-length column, and also affects trailing-space removal. CHAR is shorthand for CHARACTER. NATIONAL CHAR (or its equivalent short form, NCHAR) is the standard SQL way to define that a CHAR column should use the default character set. This is the default in MySQL. The BINARY attribute causes sorting and comparisons to be case sensitive. From MySQL 4.1.0 on, the ASCII attribute can be specified. It assigns the latin1 character set to a CHAR column. From MySQL 4.1.1 on, the UNICODE attribute can be specified. It assigns the ucs2 character set to a CHAR column. MySQL allows you to create a column of type CHAR(0). This is mainly useful when you have to be compliant with some old applications that depend on the existence of a column but that do not actually use the value. This is also quite nice when you need a column that can take only two values: A CHAR(0) column that is not defined as NOT NULL occupies only one bit and can take only the values NULL and "" (the empty string).

CHAR

This is a synonym for CHAR(1).

[NATIONAL] VARCHAR(M) [BINARY]

A variable-length string. M represents the maximum column length. The range of M is 0 to 255 characters (1 to 255 prior to MySQL 4.0.2). Note: Trailing spaces are removed when VARCHAR values are stored, which differs from the standard SQL specification. From MySQL 4.1.0 on, a VARCHAR column with a length specification greater than 255 is converted to the smallest TEXT type that can hold values of the given length. For example, VARCHAR(500) is converted to TEXT, and VARCHAR(200000) is converted to MEDIUMTEXT. This is a compatibility feature. However, this conversion affects trailing-space removal. VARCHAR is shorthand for CHARACTER VARYING. The BINARY attribute causes sorting and comparisons to be case sensitive.

ENUM('value1','value2',...)

An enumeration. A string object that can have only one value, chosen from the list of values 'value1', 'value2', ..., NULL or the special " error value. An ENUM column can have a maximum of 65,535 distinct values. ENUM values are represented internally as integers.

SET('value1','value2',...)

A set. A string object that can have zero or more values, each of which must be chosen from the list of values 'value1', 'value2', ... A SET column can have a maximum of 64 members. SET values are represented internally as integers.

(MySQL also provides BLOB and TEXT family of types that can be used for holding large amounts of binary or text data, respectively. In fact you can put up to 4GB of data in a LONGBLOB or a LONGTEXT but it's an unusual application that would need to.)

- - -

You generally use varchar or char fields for storing strings that you can reasonably expect to be no more than a modest length, such as phone numbers, emails and addresses. If you need to store text of up to 65,535 characters you should use TEXT.

The documentation advises:

"For the most efficient use of storage, try to use the most precise type in all cases. For example, if an integer column will be used for values in the range from 1 to 99999, MEDIUMINT UNSIGNED is the best type. Of the types that represent all the required values, it uses the least amount of storage.

"Accurate representation of monetary values is a common problem. In MySQL, you should use the DECIMAL type. This is stored as a string, so no loss of accuracy should occur[...]"

- - -

With MySQL you can specify the AUTO_INCREMENT attribute for a numeric column at creation time so that you can automatically generate unique id numbers for newly inserted rows. As we will see in due course, it is often desirable for a table to have unique numeric identifiers for each row -- identifiers that have no other meaning or purpose other than record identification.

When you CREATE a column you can also specify whether or not NULL values are allowed. In general, NULL is allowed unless you specify NOT NULL. NOT NULL is desirable wherever possible for performance reasons, but sometimes you need NULL values. The official manual teaches us:

"The concept of the NULL value is a common source of confusion for newcomers to SQL, who often think that NULL is the same thing as an empty string. This is not the case. For example, the following statements are completely different:

```
mysql> INSERT INTO my_table (phone) VALUES (NULL);
mysql> INSERT INTO my_table (phone) VALUES ('');
```

"Both statements insert a value into the phone column, but the first inserts a NULL value and the second inserts an empty string. The meaning of the first can be regarded as "phone number is not known" and the meaning of the second can be regarded as "the person is known to have no phone, and thus no phone number."

Another real world example is a table where you record the date and the time of some event. Suppose there are cases where you know the date, but the time is either unknown or not of interest. That's an argument for storing them in two separate columns -- one date and one time -- rather than in one datetime column. Further, you will want NULL as a possibility for the time column. That's because a time value of "0" or "" will translate into "00:00:00," and later, when your application is reading that record, it will think you meant midnight when in fact you meant either "we don't know" or "we don't care."

Indexes

Another important feature of MySQL and other database servers is *indexing*. Creating indexes on columns that are frequently used as search criteria results in large speed gains in read operations as compared to tables with no indexes at all. Internally, they allow MySQL to locate rows without having to scan the entire table from beginning to end -- which gets expensive when you have a million rows. The cost is that the indexes themselves take up disk space and slow down write operations. But if used judiciously, indexes provide speed benefits that well outweigh the costs.

You can create a regular (non-unique index) on columns whose values you know might be repeated in the table.

You can specify an index as UNIQUE -- that makes it impossible to insert duplicate rows with the same value(s) in the column(s) on which the index is created, and mysql returns an error if you try.

You can also specify a column as PRIMARY KEY -- nearly identical to UNIQUE index except that a table can have only one primary key.

Having now sat through this gripping introduction to MySQL's data types and column indexing, you are ready to appreciate a CREATE TABLE statement in all its glory. We've already created the database. Now we decide to create a table called student (We might well want more columns for phone, address, etc., but we can always add them later):

```
CREATE TABLE `student` (
  `id` int(10) unsigned NOT NULL auto_increment,
  `lastname` varchar(36) NOT NULL default '',
  `firstname` varchar(36) NOT NULL default '',
  `middlename` varchar(36) NOT NULL default '',
  `email` varchar(50) NOT NULL default '',
  `zip` varchar(10) NOT NULL default '',
```

```
PRIMARY KEY (`id`)  
) ;
```

Connect to your mysql server and enter the above statement correctly, and you will see something like

```
Query OK, 0 rows affected (0.00 sec)
```

Congratulations, you have created a MySQL data table.

Fortunately you don't have to manually type or copy-and-paste SQL commands at the mysql command line. You can store any number of SQL statement in a file, start the client and feed the SQL to the server in one shot. For example, if the above statement were in a file called `create_student_table.sql`, you could say

```
mysql -username -ppassword php_university < create_student_table.sql
```

and that would be sufficient to create the table. (You can also ask mysql to read in a file full of SQL statements after you are connected and sitting at the mysql command prompt, by saying `\. file_name` (backslash dot file_name)).

With SQL, case and whitespace are not significant. You can (and should) use whitespace to make long SQL statements readable. With MySQL, case is not significant for column names. For table names, it depends on the underlying operating system. It's wise just to assume case-sensitivity so you won't get bitten.

Note that the semicolon is required when you are at the interactive command line, but with the PHP API it is a syntax error.

The next natural question might be, how do I insert new rows? Answer: with an INSERT statement. There is more than one type legal INSERT syntax. One is

```
INSERT INTO tbl_name (col1,col2) VALUES(val1,val2);
```

If I wanted to start populating the student table we created above, I might say:

```
mysql> INSERT into student (firstname, lastname, middlename, email, zip)  
-> VALUES ('David','Mintz','','dmintz@davidmintz.org','07302');
```

```
Query OK, 1 row affected (0.02 sec)
```

You might wonder what about the id column. The answer is that with columns of the `AUTO_INCREMENT` type, if you do not set it explicitly, or if you set it to `NULL`, the value gets incremented automatically to the next available number.

Let's suppose we'd like to see for ourselves and read some data from the table. You do that with a `SELECT` statement. There is a ton of fancy things you can do with `SELECT`, but in its simplest form the syntax is

```
SELECT select_expression FROM table [WHERE condition].
```

select_expression is typically the name(s) of the table column(s) that you want to fetch. You can use the wildcard `*` to mean all of the columns in the table.

```
mysql> SELECT * from student;
```

id	lastname	firstname	middlename	email	zip
1	Mintz	David		dmintz@davidmintz.org	07302

Notice that the id got set to 1. You use a WHERE clause to select a subset of rows.

```
mysql> SELECT * from student where lastname='Mintz' ;
```

id	lastname	firstname	middlename	email	zip
1	Mintz	David		dmintz@davidmintz.org	07302

If you want to change a record, use UPDATE. Again, you can limit the update to a subset of records by qualifying it with a WHERE clause. The update syntax looks like

```
mysql> UPDATE student SET middlename='E.' WHERE id = 1;
```

```
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

If you accidentally omit the WHERE clause, too bad: you have changed everybody's middle name to E.

Finally, suppose you want to delete a row. Use a DELETE statement.

```
mysql> delete from student where lastname = 'mintz' ;
Query OK, 1 row affected (0.00 sec)
```

Again, if you mean DELETE FROM *sometable* WHERE *something*, you have to say WHERE *something* or you will blow away every row in the table. MySQL doesn't feel your pain and will not warn you.

That is the barest essence of SQL: CREATE, INSERT, SELECT, UPDATE, and DELETE.

Quitting

To end a mysql session, you can type quit or \q. To get some help (not with MySQL, but with how to use the client mysql), type help.

More details

The LIKE operator is often used in a WHERE clause for simple pattern matching. Underscore () means match exactly one character. A percent sign (%) means match any number of characters, including zero.

```
mysql> select lastname, firstname from student where lastname like 'Mi%';
```

```

+-----+-----+
| lastname | firstname |
+-----+-----+
| Misch-Blum | Gisela |
| Mintz | David |
| Miranda | Carlos |
+-----+-----+
3 rows in set (0.01 sec)

```

With an ORDER BY clause you can tell MySQL how to sort the results. By default the sort order is ascending; you can reverse the sort order with the keyword DESC (i.e., ORDER BY *column_name* DESC) You can sort on multiple columns, and mix sort directions. You have to SELECT the column that you're sorting by.

```

mysql> SELECT name, species, birth FROM pet
-> ORDER BY species, birth DESC;
+-----+-----+-----+
| name | species | birth |
+-----+-----+-----+
| Chirpy | bird | 1998-09-11 |
| Whistler | bird | 1997-12-09 |
| Claws | cat | 1994-03-17 |
| Fluffy | cat | 1993-02-04 |
| Fang | dog | 1990-08-27 |
| Bowser | dog | 1989-08-31 |
| Buffy | dog | 1989-05-13 |
| Puffball | hamster | 1999-03-30 |
| Slim | snake | 1996-04-29 |
+-----+-----+-----+

```

You can also add a LIMIT clause to limit the size of the result set.

```

mysql> SELECT lastname, firstname FROM student WHERE lastname like 'd%' ORDER
BY lastname, firstname LIMIT 10;
+-----+-----+
| lastname | firstname |
+-----+-----+
| Daigneault | Marie José |
| Dajnowski | Esther |
| Danford | Natalie |
| Danilyants | Mary |
| Daxland | Gloria |
| De Caumont | Marie |
| de Jager | Marjolijn |
| Dechereux | Maurice |
| Deferrari | Matilde |
| Delgado | Arturo |
+-----+-----+
10 rows in set (0.01 sec)

```

Of course, MySQL has a wealth of functions for string manipulation, mathematical calculations, date/time manipulation, etc. Here are just a couple of cute SQL tricks.

Counting the number of rows in a table

The count() function tells you how many rows there are. Optional WHERE clause tells you how many that meet the WHERE condition. Suppose you want to find out how many students there are whose names begin with D.

```
mysql> select count(*) from student where lastname like 'd%';
+-----+
| count(*) |
+-----+
|          21 |
+-----+
1 row in set (0.03 sec)
```

COUNT() is part of a family of functions called aggregate functions, which includes others like SUM() and AVG() for computing sums and averages, respectively, of numeric columns. You can combine aggregate functions with a GROUP BY clause to generate instant statistical summary reports. Let's say I want to know the top five zip codes where the most students of PHP University reside:

```
mysql> SELECT zip, count(*) 'number of students' FROM student GROUP BY zip
ORDER BY 'number of students' DESC LIMIT 5;
+-----+-----+
| zip    | number of students |
+-----+-----+
| 10027  | 8                  |
| 10025  | 7                  |
| 10021  | 7                  |
| 10017  | 6                  |
| 07030  | 6                  |
+-----+-----+
5 rows in set (0.55 sec)
```

Notice also how we used ORDER BY *column_name* and LIMIT *number* in the above example.

We introduce another detail into the above example: *column aliasing*. With the syntax you see above, you can change the name of a column in your result set on the fly. You will need this feature to disambiguate column names if you do a JOIN that SELECTs same-named columns from two different tables -- but we will talk about JOIN next week.

For more information about MySQL aggregate functions, see <http://dev.mysql.com/doc/mysql/en/GROUP-BY-Functions.html>

Using MySQL as a calculator

You can SELECT any valid expression; it does not have to be some *_column* from some *_table*. Suppose you want to know the sales tax (in New Jersey) on \$122.13

```
mysql> select 112.13 * .06;
+-----+
| 112.13 * .06 |
+-----+
|          6.73 |
+-----+
1 row in set (0.01 sec)
```

You can also SELECT the return value of one of the MySQL data/time functions. Suppose you are too lazy to look at your digital watch to find out the current date and time, but you do happen to be sitting in front of the MySQL command line connected to a server in your timezone:

```
mysql> select now();
+-----+
```

```
| now() |
+-----+
| 2004-10-26 11:01:59 |
+-----+
1 row in set (0.00 sec)
```

In a PHP/MySQL application, you sometimes have a choice whether to make MySQL or PHP perform certain kinds of work. It would be ridiculously inefficient to establish a connection to your MySQL database just to get the current time or to do simple arithmetic, but there are certainly situations where the power of MySQL is helpful. For example, it is more efficient to use ORDER BY and let MySQL sort your results than to put all the results in a PHP array and then sort it yourself.

Investigating tables, and other tips

Type `show tables` to see a list of table in the currently selected database.

Type `describe table_name` to see a description of the columns and data types in the table named `table_name`.

Type `\P` to turn on paging so your output doesn't fly off the screen.

Terminate a statement with `\G` instead of `;` to get vertical format of your output. Works nicely with `describe table_name` and `\P` when `table_name` has a lot of columns.

Removing tables and databases

Deleting tables and databases with MySQL is (perhaps too) easy. Say `DROP DATABASE database_name` or `DROP TABLE table_name`. Poof.

Creating a MySQL User

MySQL has a sophisticated security system of users and privileges that allows fine-grained control of who is allowed to do what with which databases and tables, and what hosts the users can log in from. That information itself is stored, appropriately enough, in a MySQL database called `mysql`. MySQL gurus recommend using MySQL GRANT and REVOKE statements for managing users, rather than accessing the privilege tables directly.

Thus far we have created a database called `php_university` and created a table called `student`. We should now create a `mysql` user account that is privileged enough to create, drop, and alter tables and select, update, insert and delete rows in any table in that database:

```
mysql> GRANT ALL ON php_university.* to 'php'@'localhost' IDENTIFIED BY
'phis2kool';
Query OK, 0 rows affected (0.00 sec)

mysql> flush privileges;
Query OK, 0 rows affected (0.00 sec)
```

With the GRANT statement above, we created a new user named `php` who could do just about anything with the database `php_university`; we specified that user `php` could only log in from

localhost; and we gave the user password `phpis2kool`. Then we told `mysqld` to reload the privilege tables so the change would take effect. You should now be able to quit and log back in as user `php` by typing

```
mysql -uphp -pphis2kool php_university
```

If that works, congratulations. Your MySQL setup is in working order and you're ready to start talking to it programmatically via PHP. (If it doesn't, try again from scratch with extreme attention to every step, saving every command and error message in a text file. Then paste the text into an email and post it to the class forum with a verbose description of your system setup: operating system, what version of everything, etc.)

Talking to MySQL via PHP

If you read about database access in *Learning PHP 5* (Sklar) you will notice that he's using the PEAR/DB class for database access, whereas in this discussion we are using native PHP `mysql` functions. We will work with the PEAR/DB interface later.

PHP is justly famous for how easy it makes the creation of web pages that can access databases.

You also have a number of design choices with respect to your general approach. You can either use an *abstraction layer* such as the aforementioned PEAR/DB, or use native PHP functions. In the former case, your code does not talk directly to the database using built-in (native) PHP functions; instead, you use the interface provided by the abstraction layer, which in turn talks to the database using the appropriate native PHP API. The advantages of doing this are twofold: (1) it makes it *much* easier to port your application to a different database server at some future time, and (2) abstraction layers usually provide a wealth of convenience functions to make your programming life easier, thereby reducing development time. The disadvantages are (1) the abstraction layer will make your code run slower, and (2) the abstraction layer may not support some of your chosen databases' more esoteric and sophisticated features.

Introducing `mysqli` – the new improved PHP interface to MySQL

The aging PHP `mysql` extension has performed admirably since the mid 1990's, but increasing compatibility issues, among other problems, have prompted a developer named Georg Richter to step up and write the new `mysqli` extension for MySQL 4.1 and PHP 5. If you have any choice in the matter, you should *use the new `mysqli` rather than its predecessor*.

`mysqli` provides both an object-oriented (OO) and a procedural interface. For now, we will investigate the latter, since we have not yet introduced OO PHP. (The procedural interface itself uses objects as return values and arguments, but does not require you to use OO syntax to work with these objects.)

Doing SELECT queries

Step 1: Connect to your database using the `mysqli_connect()`.

[Per the PHP documentation:]

`mysqli` **mysqli_connect** ([string host [, string username [, string passwd [, string dbname [, int port [, string socket]]]]]])

Returns a **mysqli** object which represents the connection to a MySQL server, or **FALSE** if the connection failed.

The arguments you will usually be concerned with are the first four: host, username, password, dbname. Assuming a database called `php_university` and a user account on the localhost with username `php` and password `phpis2kool`, you could get a connection (`$link`) to that database with:

```
$link = @mysqli_connect('localhost', 'php', 'phpis2kool', 'php_university')
      or die("Could not connect to database: " . mysqli_connect_error()) ;
```

Note that in our example we die with an error message if the connection fails. As you may have inferred, `mysqli_connect_error()` returns a string containing the error message that MySQL gives you in the event of a failed connection attempt. We opted to use the `@` error suppression operator to suppress an error message that might otherwise be emitted..

Step 2: Execute a query against the database

mixed **mysqli_query** (*mysqli* link, *string* query [, *int* resultmode])

For *SELECT*, *SHOW*, *DESCRIBE* or *EXPLAIN* **mysqli_query()** will return a result object. Otherwise, returns true on success or false on failure. (The third argument, *resultmode*, pertains to executing multiple queries in one call, and is beyond the scope of this introduction.)

Assuming a table in our database called `student` that has columns called `lastname` and `firstname`, we can get the names of those whose last names begin with the letter M like so:

```
$query = "SELECT firstname, lastname FROM student
         WHERE lastname LIKE 'M%'
         ORDER BY lastname";
$result = @mysqli_query($link, $query) or die(mysqli_error($link));
```

`$result` is a *mysqli_result* object; this is the thing through which you get your data.

Step 3: Display the results

You will often want to iterate through the results in a `while()` loop, displaying each row of data. In pseudocode:

```
while (there is another row of $data in $result) { display $data }
```

That data can come in any of several forms, at your pleasure: as an associative (string-indexed) array, in which the column names are the (string) keys; as a numerically indexed array, in which the elements are ordered in the order you specified the columns in your query; as both a numeric and an associative array, in which each field is available by its string key (column name) and by its position in the array; or as an object, in which each field is a property. Fetching data into associative arrays is often convenient, since it relieves you of the burden of remembering the order of the columns. One way to do this is with `mysqli_fetch_assoc($result)`¹:

¹ See <http://php.net/manual/en/ref.mysqli.php> for information about the various `mysqli_fetch_XXX` functions.

```
while ($student = mysqli_fetch_assoc($result)) {
    echo htmlspecialchars("$student[lastname], $student[firstname]")."<br />";
}
```

However, it could happen that the query returns 0 results. It is wise to check before displaying. You might also want to report the number of matching records to the user. For that we have `mysqli_num_rows()`. It takes a `mysqli_result` object as argument, and returns the size of that results set.² :

```
$count = mysqli_num_rows($result);
```

Step 4: Free the result and close the connection

When you are finished using the result set, you should explicitly free the memory it uses so as to return resources to PHP and MySQL as soon as possible. This happens automatically when the script terminates, but it is good practice to do it explicitly yourself anyway.

```
mysqli_free_result($result);
```

The same is true of the database connection: let it go when you are through with it.

```
mysqli_close($link);
```

Recap: Putting it all together

You should now be able to follow the following short PHP script, which fetches and displays all the first and last names of all the hypothetical PHP University students whose last names begin with M.

```
<?php
$link = @mysqli_connect('localhost','php','phpis2kool','php_university')
        or die("Could not connect to database: " . mysqli_connect_error());

$query = "SELECT firstname, lastname FROM student
        WHERE lastname LIKE 'M%'
        ORDER BY lastname";
$result = @mysqli_query($link, $query) or die(mysqli_error($link));

$count = mysqli_num_rows($result);

echo "Results found: $count<br />" ;

if ($count) {
    while ($student = mysqli_fetch_assoc($result)) {
        echo htmlspecialchars(
            "$student[lastname], $student[firstname]")."<br />";
    }
}
mysqli_free_result($result);
mysqli_close($link);
?>
```

² Except when you are using unbuffered queries, which you should do when you expect quite a large result set and do not want to consume too much memory. But that's beyond the scope of our discussion.

Escaping String Data

In the above example, the search term (“M”) was hard-coded. In real PHP-life, things like search terms and input data for updates and deletes will often be PHP variables holding data entered by your users. Consider a form that provides a search interface to our PHP University student directory. It might well have a textfield for the user to enter a last name, which would become part of the SQL query:

```
$sql = “SELECT * FROM student WHERE lastname = ‘$_GET[lastname]’ ”
```

If `$_GET['lastname']` happens to be something like “O’Reilly”, then `$sql` would be

```
“SELECT * FROM student WHERE lastname = 'O'Reilly' ”
```

The query would fail, because MySQL would think the lastname was 'O' and the stuff after it, a syntax error. In addition to subjecting user input to your usual rigorous validation, you also have to *escape any embedded quotes inside strings* that you plan to use in your SQL queries.

Naturally, the `mysqli` extension provides more than one way to do it. One way is with *placeholders* and *bound parameters*. We will study these in Session Seven. For now we will use another technique: escape our strings with

```
string mysqli_escape_string ( mysqli link, string escapestr )
```

Revising the above example, we have

```
$lastname = mysqli_escape_string($link, $_GET['lastname']);  
$sql = “SELECT * FROM student WHERE lastname = '$lastname' ”;
```

Now `$sql` would expand to

```
“SELECT * FROM student WHERE lastname = 'O\'Reilly' ”
```

Doing INSERT Queries

For demonstration purposes in the following example, our data comes from `$data`. In actual life it will often be POST data from an HTML form. The procedure is straightforward: escape your data, connect, run the query, disconnect.

`mysqli_affected_rows()` returns a the number of rows affected by the last operation you did, whatever it was. Sometimes it may be zero without being an error.

```
$data = array('lastname' => "O'Reilly", 'firstname' => 'Sean', 'email' =>  
'escapable@example.net', );  
  
$link = mysqli_connect("localhost", "php", 'phpis2kool', "php_university");  
  
if (!$link) {  
    printf("Connect failed: %s\n", mysqli_connect_error());  
    exit();  
}  
  
foreach ($data as $name => $value) {  
    $data[$name] = mysqli_escape_string($link, $value);  
}
```

```

$query = "INSERT INTO student (lastname, firstname, email)
        VALUES ('$data[lastname]','$data[firstname]','$data[email]')";

// for debugging
echo "$query<br />";

// insert or die
if (! $result = mysqli_query($link,$query)) {
    die(mysqli_error($link));
}

// report the outcome

printf( "Rows inserted: %d<br />",  mysqli_affected_rows($link));

// the student table has an auto-incrementing id column,
// mysqli_insert_id() returns the auto-generated id from
// the last INSERT operation

printf("Record id: %d<br />", mysqli_insert_id($link));
mysqli_close($link);

```

Doing UPDATE Queries

UPDATE syntax is similar to INSERT. You just want to make sure you have a WHERE clause, unless you really intend to update all the rows.

Assume again you have an array of data, including the record id, and a mysqli connection object in \$link:

```

$data = array('id' => 2350, 'lastname' => "O'Reilly",'firstname' =>
'Sean','email' => 'sean@example.org',);

$fields = array_keys($data);
foreach ($fields as $field) {
    $data[$field] = mysqli_escape_string($link, $data[$field]);
}

$query = "UPDATE student
        SET email = '$data[email]', lastname = '$data[lastname]',
            firstname='$data[firstname]'
        WHERE id = $data[id]";

if (! $result = mysqli_query($link,$query)) { die(mysqli_error($link)); }

echo mysqli_affected_rows($link) . " rows updated<br />" ;

```

Doing DELETE Queries

```

$query = "DELETE from student WHERE id = 2350"
$result = mysqli_query($link,$query);
echo mysql_affected_rows($link) . " row deleted<br />";

```

Even More Details: Formatting raw mysql timestamp values

MySQL timestamp values before MySQL 4.1 are stored in this format: YYYYMMDDHHMMSS. Not very legible.

```
mysql> select * from guestbook_comment limit 1;
+-----+-----+-----+
| posted_by | posted_on | text |
+-----+-----+-----+
| Mr. Bludgeon | 20041027163210 | this is a comment |
+-----+-----+-----+
1 row in set (0.00 sec)
```

After 4.1, the default format is more legible, but still not as human-friendly as you might like:

```
mysql> select * from guestbook_comment limit 1;
+-----+-----+-----+
| posted_by | posted_on | text |
+-----+-----+-----+
| Mr. Bludgeon | 2004-10-27 23:31:40 | this is a comment |
+-----+-----+-----+
1 row in set (0.00 sec)
```

You could parse and re-format it with PHP, but it's easier to use MySQL's `date_format()` function. The sole downside is that you have yet another set of formatting symbols to deal with. The list is available at http://dev.mysql.com/doc/mysql/en/Date_and_time_functions.html under the section about the `date_format()` function.

```
mysql> select date_format(posted_on,"%a %d-%b-%Y %l:%i %p") 'posted_on' from
guestbook_comment limit 1;
+-----+
| posted_on |
+-----+
| Wed 27-Oct-2004 4:32 PM |
+-----+
1 row in set (0.00 sec)
```

Assignment

We will go back to the same `guestbook.inc` from the guestbook application we played with for Week 4 with a few revisions, and make it use a MySQL database as a backend instead of a text file. (Please: follow slavishly the following instructions regarding table structure, username and password -- i.e, repress your natural creativity -- so I will be able to drop your code right into my own setup and run it without having to adjust anything.)

Connect to your mysql server from the command line.

Create a database called `php_university`.

Create a table in that database called `guestbook_comment` using the following SQL statement:

```
CREATE TABLE `guestbook_comment` (
  `posted_by` varchar(30) NOT NULL default '',
  `posted_on` timestamp(14) NOT NULL,
  `text` varchar(255) NOT NULL default '',
  `comment_type` enum('praise','rant','question','other') NOT NULL
  default 'other',
  `id` smallint(5) unsigned NOT NULL auto_increment,
  PRIMARY KEY (`id`),
  UNIQUE KEY `poster_text_idx` (`posted_by`,`text`),
  KEY `posted_on_idx` (`posted_on`)
) ;
```

Create a user account called guest with password splatt78 by running the following commands:

```
GRANT SELECT,INSERT on php_university.guestbook_comment TO
'guest'@'localhost' IDENTIFIED BY 'splatt78';
FLUSH PRIVILEGES;
```

Download http://davidmintz.org/php_course/6/assignment.zip and unzip it. It contains the necessary SQL scripts and stub code for the assignment.

In guestbook.inc:

Write a function called `db_connect()` that establishes a connection to your MySQL server's `php_university` database. If it succeeds, return the link; otherwise, return `false`. Put this function in `guestbook.inc`. Use the error suppression operator `@` in your database connection code.

Rewrite the `display_comments()` function so that it fetches the last 10 comments (rows) from the guestbook (`guestbook_comments`) by descending order of date, and displays them to the browser. `display_comments()` will take one argument: a `mysqli` database connection object.

Implement logic that follows the following procedure:

1. execute the appropriate `SELECT` query, or return `false` if it fails
2. if the result set is empty, echo "There are no comments in the guestbook" and return `true`
3. otherwise: iterate through the result set with a `while` loop printing the data and time of each post, the type of comment (rant | praise | question | other), and the comment text. Format the metadata (date, time and poster) as italic text.
4. After this loop, return `true`.

Rewrite the `save_comment()` function so that it follows a pattern similar to the above. This function takes two arguments: (1) an array holding the data for the guestbook entry, and (2) the database connection object.

1. iterate through the array that the function receives as argument calling `mysqli_escape_string()` on each of the values of the array. TIP: if you use `foreach` for this, use this syntax:

```
foreach ($data as $key => $value) {
    $data[$key] = mysqli_escape_string($link,$value);
}
```

2. execute the appropriate `SQL INSERT` query, or return `false` if it fails
3. return the return value of `mysqli_affected_rows()`

TIP 1: Get an early start on this one. If you wait until x minus a few hours and run into trouble connecting to your database or some such, I'm not likely to have enough time to help you sort it out.

TIP 2: Play around with all your `SQL` queries through the command line interface and get them working before you try running them from `PHP`.

TIP 3: Use the `date_format()` snippet shown above to convert the timestamp into a more human-friendly format.

TIP 4: In your SELECT statement, you will discover that the formatted date strings get sorted alphabetically rather than in chronologically descending order. A trick that works is to SELECT both 'posted_on' as is, as well as `date_format(posted_on, "%a %d-%b-%Y %l:%i %p")` 'date' and ORDER BY 'posted on' but *display* the column aliased as 'date'.

Do not change any of the logic in `guestbook.php`. In fact you need not add or change anything except the three function bodies in `guestbook.inc` -- unless you want to embellish the presentation for your own amusement. Please upload two files: `guestbook.inc` and `guestbook.php`

Reading

<http://www.mysql.com/doc/en/index.html> (Learn to love it!)

<http://www.analysisandsolutions.com/code/mybasic.htm>

<http://www.php.net/manual/en/ref.mysql.php> etc

28-Mar-2005 09:32 am