SEMESTER – VI - III BSc IT

18BIT61C – Programming in PHP

UNIT V: Database & MySQL – Installing MySQL – Integrating PHP & MySQL – Connecting to MySQL – MySQL Queries – Dataset – Multiple Connection – Error Checking – Creating MySQL Database with PHP – MySQL Data types – MySQL Functions.

TEXT BOOKS

1. Steve Suehring, Tim Converse, Joyce Park , "PHP 6 and MySQL 6 Bible", Wiley India pvt. Ltd., Edition, 2009 (Unit – V).

REFERENCE BOOKS

1. Matt Doyle, " Beginning PHP 5.3", Wiley India pvt. Ltd, First edition, 2010.
2. Luke welling and Laura Thomson, "PHP and MySQL Web Development", 5th Edition, 2016.

5. Introducing Databases and MySQL

Databases and PHP go together like cake and ice cream, the single greatest advantage of PHP over similar products is the unsurpassed choice and eases of database connectivity it. PHP supports native connections to a number of the most popular database server types, open source and commercial alike. Almost any database that will open its application programming interface (API) to the public seems to be included eventually. For any unsupported databases, there's generic ODBC (Open Database Connectivity) support.

What Is a Database?

A *database* is a collection of data. The term database usually indicates that the collection of data is stored on a computer.

Databases implemented through a computer are created within software. That software, commonly known as a database application, controls how the actual data is stored and retrieved. Some database applications include Microsoft Access and OpenOffice.org's Base. Sometimes, databases are stored in a central location and managed by a database server. A database server is a database application built with multiple users in mind.

Most of the time when programming PHP you'll be accessing a database server. Some database servers include PostgreSQL, MySQL, Microsoft's SQL Server, and the Oracle suite of databases. The database servers called RDBMS, which is an acronym for relational database management system.

Database servers usually have one or more distinct APIs for programmatically creating, accessing, managing, searching, and replicating the data they hold. It is through the API that you connect to and work with data stored in database servers when using PHP.

There is no requirement that an RDBMS be used to store data. Other data stores can be used such as a flat file or a table known as a hash table. These are perfectly fine for some applications, especially smaller applications; however, for larger applications or applications that require optimal speed for large data stores, an RDBMS is a requirement.

Why a Database?

Advantages of using a database instead of static pages or included text files are

- Maintainability and scalability
- Portability
- Avoiding awkward programming

- Searching

PHP-Supported Databases

PHP Data Objects (PDO) was introduced back with the 5.1 release of PHP. PDO creates a consistent, abstracted interface to database servers and data. PHP offers several database-specific drivers for both PDO and non-PDO access. The PHP web site contains a list with the latest information about databases that can be integrated along with the PDO abstraction layer and other abstraction layers.

See **www.php.net/pdo** for more information.

MySQL

MySQL, (officially pronounced my- S - Q - L and not "mysequel"), is an incredibly popular and powerful RDBMS. MySQL provides one of the letters in the ubiquitous acronym "LAMP," which is an abbreviation for Linux, Apache, MySQL, PHP/Perl/Python.

MySQL has become so popular for several reasons.

1. MySQL is free (as in price), although the licensing has changed.
2. MySQL is also stable, meaning that it's not prone to crashing even under load.
3. MySQL is lightweight, meaning that it doesn't require many resources to install or run.
4. MySQL is fast and easy to use.
5. MySQL is powerful, with all of the features required for web applications.

MySQL AB, which is the company behind MySQL (owned by Sun), changed the licensing for MySQL relatively recently. In the latest iteration as of this writing, MySQL offers a product called MySQL Server Community Edition, which is essentially the same as the MySQL Enterprise Server, but is lacking official MySQL support and some graphical user interface (GUI) tools. MySQL AB's support is excellent; Otherwise, the MySQL Server Community Edition is your choice.

For more information on the differences between the two versions,

see **www.mysql.com/products/which-edition.html**.

Installing MySQL

MySQL's database server can be downloaded from MySQL's web site at **www.mysql.com** or download section for MySQL is currently located at **http://dev.mysql.com/downloads.** However, realize that most distributions of Linux include their own MySQL server package.

Installing MySQL on Linux

There are several distributions upon which you might find yourself installing MySQL.

Installing MySQL Server on Debian and Ubuntu

Debian's dpkg and apt installation and package management tools make installation of MySQL incredibly easy. Debian is a system administrator's dream because it's so stable, package installation is so easy, and the packages are maintained and configured with excellent defaults. But enough evangelizing; installation of MySQL server on Debian requires superuser privileges and is accomplished simply by running apt-get:

apt-get install mysql-server

Of course, that assumes that you have correctly configured sources in /etc/apt/sources.list. For more information on APT and configuration of the sources.list file, see www.debian.org/doc/manuals/apt-howto/ch-basico.en.html.

Debian's package management system will install and configure any necessary prerequisites for you. Debian separates MySQL into its components such as server, client, and libraries. Therefore, in order to use MySQL and PHP together, you should install the php5-mysql package:

apt-get install php5-mysql

As you can see by that installation command, the PHP5 version of the interface is being installed.

Finally, you'll likely also want to install the MySQL command-line interface (CLI), which is accomplished by installing the mysql-client package:

apt-get install mysql-client

MySQL will now be installed and ready to use on your Debian server. However, by default the MySQL server won't listen on anything by localhost. To change this, edit

/etc/mysql/my.cnf

and comment out the skip-networking line with a pound sign or hash mark (#), so it looks like this:

#skip-networking

Now restart the MySQL server by typing this command:

/etc/init.d/mysql restart

Installing MySQL on Microsoft Windows

Default installation on any version of Windows is now much easier than it used to be, as MySQL now comes neatly packaged with a native Windows installer. Simply download the installer package, usually an msi, and run it. This will walk you through the trivial process and by default will install everything under C:\Program Files\MySQL, which is probably as good a place as any.

The MySQL installer will attempt to install itself as a service, which means you need Administrator rights on the computer upon which MySQL is being installed. Part of the installation process will configure the MySQL server. During this portion of the installation, you can configure things like the root password, the port on which MySQL will listen, and whether to include the MySQL utilities in the Windows path. The Windows install is now so simplified that for most cases you can simply click "Next" to continue and, where you have an exception, refer to the online manual for MySQL at www.mysql.com.

Integrating PHP and MySQL

After you've installed and set up your MySQL database, you can begin to write PHP scripts that interact with it.

Connecting to MySQL

The basic command to initiate a MySQL connection is

```
mysql_connect($hostname, $user, $password);
```

if you're using variables, or

```
mysql_connect('localhost', 'root', 'sesame');
```

if you're using literal strings,

The password is optional, depending on whether this particular database user requires one. If not, just leave that variable off. You can also specify a port and socket for the server ($hostname:port:socket), but unless you've specifically chosen a nonstandard port and socket, there's little to gain by doing so.

The corresponding mysqli function is mysqli_connect, which adds a fourth parameter allowing you to select a database in the same function you use to connect. The function mysqli_select_db exists, but you'll need it only if you want to use multiple databases on the same connection.

You do not need to establish a new connection each time you want to query the database in the same script. You will need to run this function again, however, for each script that interacts with the database in some fashion.

Next, you'll want to choose a database to work on:

```
mysql_select_db($database);
```

if you're using variables, or

```
mysql_select_db('phpbook');
```

if you're using a literal string.

You must select a database each time you make a connection, which means at least once per page or every time you change databases. Otherwise, you'll get a Database not selected error. Even if you've created only one database per daemon, you must do this, because MySQL also comes with default databases (called mysql and test) you might not be taking into account.

You may find it convenient to group all your connection information into a custom connect function and put it someplace where you can access it from all your scripts, such as the php includes directory, or in the case of a virtual server, a site-specific include file. This function might look like the following:

```
// Connect to a single db
function qdbconn()
{
        $dbUser = "myuser";
        $dbPass = "mypassword";
        $dbName = "mydatabase";
        $dbHost = "myhost";
         if (!($link=mysql_connect($dbHost, $dbUser, $dbPass)))
        {
           echo error_log(mysql_error(), 3, "/tmp/phplog.err");
        }
        if (!mysql_select_db($dbName, $link))
        {
           echo error_log(mysql_error(), 3, "/tmp/phplog.err");
        }

}
```

If you like, you could extend this function by creating links (for example, $link1, $link2) to multiple databases on the same server. This code also records a MySQL error message in the PHP error log.

Now that you've established a connection to a specific database, you're ready to make a query.

Making MySQL Queries

A database query from PHP is basically a MySQL command wrapped up in a tiny PHP function called mysql_query(). This is where you use the basic SQL workhorses of SELECT, INSERT, UPDATE, and DELETE. The MySQL commands to CREATE or DROP a table can also be used with this PHP function if you do not wish to make your databases using the MySQL client. You could write a query in the simplest possible way, as follows:

mysql_query("SELECT Surname FROM personal_info WHERE ID < 10");

PHP would dutifully try to execute it. However, there are very good reasons to split up this and similar commands into two lines with extra variables, like this:

$query = "SELECT Surname FROM personal_info WHERE ID < 10";
$result = mysql_query($query);

The main rationale is that the extra variable gives you a handle on an extremely valuable piece of information. Every MySQL query gives you a receipt whether you succeed or not.

Another advantage of assigning the query string to a variable is that you can more easily view the query if you run into an error.

The function mysql_query takes as arguments the query string (which should not have a semicolon within the double quotation marks) and optionally a link identifier. Unless you have multiple connections, you don't need the link identifier. It returns TRUE (nonzero) integer values if the query was executed successfully even if no rows were *affected.* It returns a FALSE integer if the query was illegal or not properly executed for some other reason.

If you need to use multiple databases in your script, you can use code like this:

```
$query = "SELECT Surname FROM personal_info WHERE ID < 10";
$result = mysql_query($query, $link_1);
$query = "SELECT * FROM orders WHERE date > 20030702";
$result = mysql_query($query, $link_2);
```

As expected, the MySQL improved analog for this function is mysqli_query. It is very similar to its counterpart; however, the link and query parameters change places, and a third parameter allows you to specify a result flag indicating how PHP should handle the result.

If your query was an INSERT, UPDATE, DELETE, CREATE TABLE, or DROP TABLE and returned TRUE, you can now use mysql_affected_rows to see how many rows were changed by the query. This function optionally takes a link identifier, which is only necessary if you are using multiple connections.

It *does not* take the result handle as an argument! You call the function like this, without a result handle:

$affected_rows = mysql_affected_rows();

If your query was a SELECT statement, you can use mysql_num_rows($result) to find out how many rows were returned by a successful SELECT.

The mysqli_affected_rows and mysqli_num_rows behave exactly the same as their mysql_counter parts.

Fetching Data Sets

One thing that often seems to temporarily stymie new PHP users is the whole concept of fetching data from PHP. It would be logical to assume that the result of a query would be the desired data, but that is not correct. What actually happens is that a mysql_query() command pulls the data out of the database and sends a receipt back to PHP reporting on the status of the operation. At this point, the data exists in a purgatory that is immediately accessible from neither MySQL nor PHP. The data is there, but it's waiting for the commanding officer to give the order to deploy. It requires one of the mysql_fetch functions to make the data fully available to PHP.

The fetching functions are as follows:

- mysql_fetch_row: Returns row as an enumerated array
- mysql_fetch_object: Returns row as an object
- mysql_fetch_array: Returns row as an associative array
- mysql_result: Returns one cell of data

The differences among the three main fetching functions is small. The most general one is mysql_fetch_row, which can be used something like this:

```
$query = "SELECT ID, LastName, FirstName
FROM users WHERE Status = 1";
$result = mysql_query($query);
while ($name_row = mysql_fetch_row($result))
{
        print("{$name_row[0]} {$name_row[1]} {$name_row[2]}<BR>\n");
}
```

This code will output the specified rows from the database, each line containing one row or the information associated with a unique ID (if any).

The function mysql_fetch_object performs much the same task, except the row is returned as an object rather than an array. Obviously, this is helpful for those among the PHP brethren who utilize the object-oriented notation:

```
$query = "SELECT ID, LastName, FirstName
FROM users WHERE Status = 1";
$result = mysql_query($query);
while ($row = mysql_fetch_object($result))
{
        echo "{$row->ID}, {$row->LastName}, {$row->FirstName}<BR>\n";
}
```

The most useful fetching function, mysql_fetch_array, offers the choice of results as an associative or an enumerated array — or both, which is the default. This means you can refer to outputs by database field name rather than number:

```
$query = "SELECT ID, LastName, FirstName
FROM users WHERE Status = 1";
$result = mysql_query($query);
while ($row = mysql_fetch_array($result)) {
        echo "{$row['ID']}, {$row['LastName']}, {$row['FirstName']}<BR>\n";
```

```
}
```

Remember that mysql_fetch_array can *also* be used exactly the same way as mysql_fetch_row — with numerical identifiers rather than field names. If you want to specify offset or field name rather than making both available, you can do it like this:

```
$offset_row = mysql_fetch_array($result, MYSQL_NUM);
or
$associative_row = mysql_fetch_array($result, MYSQL_ASSOC);
```

It's also possible to use MYSQL_BOTH as the second value, but because that's the default, it's redundant.

The PHP junta now recommends use of mysql_fetch_array over mysql_fetch_row because it offers increased functionality and choice at little cost in terms of programming difficulty, performance loss, or maintainability.

Last and least of the fetching functions is mysql_result(). You should only even *consider* using this function in situations where you are positive you need only one piece of data to be returned from MySQL. An example of its usage is:

```
$query = "SELECT count(*) FROM personal_info";
$db_result = mysql_query($query);
$datapoint = mysql_result($db_result, 0, 0);
```

The mysql_result function takes three arguments: *result identifier*, *row identifier*, and (optionally) *field*. Field can take the value of the field offset as above or its name as in an associative array ("Surname") or its MySQL field-dot-table name ("personal_info.Surname"). Use the offset if at all possible, as it is substantially faster than the other two.

All of the PHP functions for fetching MySQL data have identical mysqli counterparts. They take the same parameters and return comparable results.

A special MySQL function can be used with any of the fetching functions to more specifically designate the row number desired. This is mysql_data_seek, which takes as arguments the result identifier and a row number and moves the internal row pointer to that row of the data set.

The most common use of this function is to reiterate through a result set from the beginning by resetting the row number to zero, similar to an array reset. This obviates another expensive database call to get data you already have sitting around on the PHP side. Here's an example of using mysql_data_seek():

```
<?php
        echo("<TABLE>\n<TR><TH>Titles</TH></TR>\n<TR>");
        $query = "SELECT title, publisher FROM books";
        $result = mysql_query($query);
        while ($book_row = mysql_fetch_array($result))
        {
                echo("<TD>$book_row[0]</TD>\n");
        }
        echo("</TR></TABLE><BR>\n");
        echo("<TABLE>\n<TR><TH>Publishers</TH></TR>\n<TR>");
        mysql_data_seek($result, 0);
        while ($book_row = mysql_fetch_array($result))
        {
                echo("<TD>{$book_row[1]}</TD>\n");
```

```
        }
        echo("</TR></TABLE><BR>\n");
?>
```

Without using mysql_data_seek, the second usage of the result set would turn back no 0 rows because it has already iterated through to the end of the dataset and the pointer stays there until you explicitly move it. This handy function helps greatly when you are formatting data in a way that does not place fields in columns and records in rows.

Getting Data about Data

You only need four PHP functions to put data into or get data out of a preexisting MySQL database:

mysql_connect, mysql_select_db, mysql_query, and mysql_fetch_array.

PHP offers extensive built-in functions to help you learn the name of the table in which your data resides, the data type handled by a particular column, or the number of the row into which you have just inserted data. With these functions, you can effectively work with a database about which you know very little.

The MySQL metadata functions fall into two major categories:

- Functions that return information about the previous operation only
- Functions that return information about the database structure in general

A very commonly used example of the first type is mysql_insert_id(), which returns the auto incremented ID assigned to a row of data you just inserted. A commonly used example of the second type is mysql_field_type(), which reveals whether a particular database field's data must be an integer, a varchar, text, or what have you. Observe however, that this function is also deceptively named.

Rather than returning the MySQL type, it returns the PHP data type. For example, an ENUM-type field will return 'string'. Use mysql_field_flags to return more specialized field information. This should be apparent when you consider that it works on a result rather than on an actual MySQL field.

Multiple Connections

Unless you have a specific reason to require multiple connections, you only need to make one database connection per PHP page.

Conversely, there's no easy way to keep your connection open from page to page - because PHP and MySQL would never know for sure when to close it after visitors wander off. Therefore, your connection is closed at the end of each script unless you use persistent connections.

The main time that you need to use different connections is when you're querying two or more completely separate databases. The most common situation in which you might do this is when you're using MySQL in a replicated situation. MySQL replication is accomplished through a master-slave setup, where you typically get reads from a slave and make writes to the master.

In this example, we are using connections from three different databases on different servers:

```
<?php
        $link1 = mysql_connect('host1', 'me', 'sesame');
        mysql_select_db('userdb', $link1);
```

```
        $query1 = "SELECT ID FROM usertable
        WHERE username = '$username'";
        $result1 = mysql_query($query1, $link1);
        $array1 = mysql_fetch_array($result1);
        $usercount = mysql_num_rows($result1);
        mysql_close($link1);
        $today = '2002-05-01';
        $link2 = mysql_connect('host2', 'myself', 'benne');
        mysql_select_db('inventorydb', $link2);

        $query2 = "SELECT sku FROM widgets
        WHERE ship_date = '$today'";
        $result2 = mysql_query($query2, $link2);
        $array2 = mysql_fetch_array($result2);
        $widgetcount = mysql_num_rows($result2);
        mysql_close($link2);
        if ($usercount > 0 && $widgetcount > 0)
        {
                $link3 = mysql_connect('host3', 'I', 'seed');
                mysql_select_db('salesdb', $link3);
                $query3 = "INSERT INTO saleslog (ID, date, userID, sku)
                VALUES (NULL, '$today', '$array1[0]', '$array2[0]')";
                $result3 = mysql_query($query3, $link3);
                $insertID = mysql_insert_id($link3);
                mysql_close($link3);
                if ($insertID >= 1)
                {
                        print("Perfect entry");
                }
                else
                {
                        print("Danger, danger, Will Robinson!");
                }
        }
        else
        {
                print("Not enough information");
        }
?>
```

In this example, we have deliberately kept the connections as discrete as possible for clarity's sake, even going to the trouble to close each link after we use it. Without the mysql_close() commands, we would be running multiple concurrent connections.

Building in Error Checking

This section could have been titled "Die, die, die!" because the main error-checking function is actually called die().

die() is not a MySQL-specific function — the PHP manual lists it in "Miscellaneous Functions." It simply terminates the script (or a delimited portion thereof) and returns a string of your choice.

```
mysql_query("SELECT * FROM mutual_funds
WHERE code = '$searchstring'")

or die("Please check your query and try again.");
```

MySQL via PHP returned very insecure and unenlightening (except to crackers) error messages upon encountering a problem with a database query. die() was often used as a way to exert control over what the public would see on failure. Now that no error messages are returned at all, die() may be even more necessary - unless you want your visitors to be left wondering what happened.

Other built-in means of error-checking are error messages. These are particularly helpful during the development and debugging phase, and they can be easily commented out in the final edit before going live on a production server. As mentioned, MySQL error messages no longer appear by default. If you want them, you have to ask for them by using the functions mysql_errno() (which returns a code number for each error type) or mysql_error() (which returns the text message). Then you can send them to a custom error log by using the error_log() function:

```
if (!mysql_select_db($bad_db))
{
        print(mysql_error());
}
```

There's more to database error handling than judicious use of die(), however. Servers become unavailable; data sets get corrupted, and so forth. We've been fairly liberal in setting up connections and executing queries, but ideally, every interaction with the database should be nested inside a conditional that returns the desired result on success and a nice clean error page on failure. This is where die() drops the ball. Execution immediately stops for the entire script, leaving off, if nothing else, closing tags for your HTML page if they are defined in PHP. Additionally, there may be plenty more perfectly good scripting or HTML left to go on the page - code that is unaffected by a dropped database connection or a failed query. .
An example of good error checking is:

```
function printError($errorMesg)
{
        printf("<B>%s </B><BR>\n", $errorMesg);
}
function notify($errorMesg)
{
        mail(webmaster@example.com, "An Error has occurred at
        example.com", $errorMesg)
}
if ($link = mysql_connect("host", "user", "pass"))
{
        // Things to do if the connection is successful
}
else
{
        printError("Sorry for the inconvenience; but we are unable
```

```
        to process your request at this time. Please check back later");
        notify("Problem connecting to database in $SCRIPT_NAME at
        line 12 on date('Y-m-D')");
}
```

Creating MySQL Databases with PHP

To create a database from PHP, the user of your scripts will need to have full CREATE/DROP privileges on MySQL. That means anyone who can get hold of your scripts can potentially blow away all your databases and their contents with the greatest of ease. This is not such a great idea from a security standpoint.

If you're even considering creating databases with PHP, do yourself a big favor and at least don't store the database username and password in a text file. Make yourself type your database username and password into a form and pass the variables to the inserting handler each and every time you use this script. This is one case where keeping the variables in an include file outside your web tree is not sufficient precaution, run the scripts manually from the command line through SSH:

```
mysql –u <username> -p <databasename> < sql-script.sql
```

For those times when you need to create databases programmatically, the relevant functions are:

- mysql_create_db(): Creates a database on the designated host, with name specified in arguments
- mysql_drop_db(): Deletes the specified database
- mysql_query(): Passes table definitions and drops in this function

A bare-bones database-generation script might look like this:

```
<?php
        $linkID = mysql_connect('localhost', 'root', 'sesame');
        mysql_create_db('new_db', $linkID);
        mysql_select_db('new_db');
        $query = "CREATE TABLE new_table (
        id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
        new_col VARCHAR(25))";
        $result = mysql_query($query);
        $axe = mysql_drop_db('new_db');
?>
```

Several other GUI tools are available that are not database-specific but will probably work with MySQL. As MySQL has become more and more popular, a number of applications for both Windows and Linux have come into play that allows you to administer MySQL databases in the graphical fashion you may have become accustomed to. However, the MySQL web site keeps a pretty comprehensive list at http://dev.mysql.com.

MySQL data types

The actual PHP functions used to create MySQL databases are trivial compared to the MySQL data structure statements that are passed in those functions. The "Database Design" has general rules on how to conceptualize a database schema and use the CREATE, DROP, and ALTER statements. To implement your abstract schema in MySQL, however, you also need to understand MySQL data types and how to use them.

The general rule is to use the smallest and most specific data type that will adequately meet the needs of this particular column in your database. MySQL is known for having compact types, such as TINYINT and TINYTEXT, that are good for things like 0/1 values or first names. It also has very large types that can store 4GB (or more) of data in one field.

There are three buckets of MySQL data types: numeric types, date and time types, and string (or character) types. For the most part, their use is fairly straightforward — in the sense that the average user is not going to know or care whether you used an INT or a MEDIUMINT. However, if you're the type of programmer who cares about doing everything in the absolutely tightest and fastest way possible, the MySQL manual gives subtle tips on maximizing efficiency — for instance, always use the DECIMAL type with money, or it takes 8 bytes to store a DATETIME but only 4 bytes to store a Unix TIMESTAMP, which PHP can convert to any date-time format you desire. Careful perusal of the "Column Types" section of the MySQL manual (at www.mysql.com/doc/en/Column_types.html) may yield hidden treasures of insight. Following table shows the current MySQL data types and their possible values. M stands for the maximum number of digits displayed, and D stands for the maximum number of decimal places in a floating-point number. Both are optional.

TBLE : MySQL Data Types

Name and Aliases	Storage size	Usage
TINYINT(M)		
BIT, BOOL, BOOLEAN are	1 byte	If unsigned, stores values from 0 to 255; otherwise, synonyms for TINYINT(1) from -128 to +127. A new Boolean type will appear in future, but until now has been implemented as a TINYINT(1).
SMALLINT(M)	2 bytes	If unsigned, stores values from 0 to 65535; otherwise, from -32768 to 32767.
MEDIUMINT(M)	3 bytes	If unsigned, stores values from 0 to 16777215; otherwise, from - 8388608 to 8388607.
INT(M)		
INTEGER(M)	4 bytes	If unsigned, stores values from 0 to 4294967295; otherwise, from -2147483648 to 2147483647.
BIGINT(M)	8 bytes	If unsigned, stores values from 0 to 18446744073709551615; otherwise, from -9223372036854775808 to 9223372036854775807. You may experience strangeness when performing arithmetic with unsigned integers of this size due to limitations in your operating system.
FLOAT(precision)	4 or 8 bytes	Where precision is an integer up to 53. If precision <= 24, converted to a FLOAT; if precision > 24 and <= 53, converted to a DOUBLE. Provided for Open DataBase Connectivity (ODBC) compatibility; in general, use the normal MySQL FLOAT and DOUBLE types.
FLOAT(M, D)	4 bytes	Single-precision floating-point number.

DOUBLE(M, D)		
DOUBLE PRECISION, REAL	8 bytes	Double-precision floating-point number.
DECIMAL(M,D)		
DEC, NUMERIC, FIXED	M+1 or M+2 bytes	An unpacked floating-point number that is stored like a CHAR. Used for small decimals, such as money.
DATE	3 bytes	Displayed in the format YYYY-MM-DD
DATETIME	8 bytes	Displayed in the format YYYY-MM-DD HH:MM:SS.
TIMESTAMP	4 bytes	Since MySQL 4.1, can no longer set display size. Displayed in the same format as DATETIME.
TIME	3 bytes	Displayed in the format HHH:MM:SS where HHH is a value from -838 to 838. This allows a TIME value to represent an elapsed time between two events.
YEAR	1 byte	Displayed in the format YYYY, which is a value from 1901 to 2155. To use an earlier date, you should use a TINYINT type.
CHAR(M)	M bytes	Fixed in length. If your string is not long enough, it will be padded with spaces at the end. M must be <= 255.
VARCHAR(M)	Up to M bytes	Variable in length. M must be <= 255.
BINARY(M)	Up to M bytes	Stores byte strings.
VARBINARY(M)	Up to M bytes	Similar to VARCHAR. Stores byte strings.
TINYBLOB or TINYTEXT	Up to 255 bytes	TINYBLOB is case-sensitive for sorting and comparison; TINYTEXT is case-insensitive.
BLOB or TEXT	Up to 64KB	BLOB is case-sensitive for sorting and comparison; TEXT is case-insensitive.
MEDIUMBLOB or MEDIUMTEXT	Up to 16MB	MEDIUMBLOB is case-sensitive for sorting and comparison; MEDIUMTEXT is case-insensitive.
LONGBLOB or LONGTEXT	Up to 4GB	LONGBLOB is case-sensitive for sorting and comparison; LONGTEXT is case-insensitive.
ENUM(value1, ...valueN)	1 or 2 bytes	Up to 65535 distinct values.
SET(value1,...valueN)	Up to 8 bytes	Up to 64 distinct values

MySQL Functions

All arguments in brackets are optional.

PHP -MySQL Functions

Function Name	Usage
mysql_affected_rows([link_id])	Use after a nonzero INSERT, UPDATE, or DELETE query to check number of rows changed.
mysql_change_user(user, *password[*, database] [, link_id])	Changes MySQL user on an open link.

mysql_close([link_id])	Closes the identified link (usually unnecessary).
mysql_connect([host][:port][:socket][, username][, password])	Opens a link on the specified host, port, socket; as specified user with password. All arguments are optional
mysql_create_db(db_name[, link_id])	Creates a new MySQL database on the host associated with the nearest open link.
mysql_data_seek(result_id, *row_num)*	Moves internal row pointer to specified row number. Use a fetching function to return data from that row.
mysql_drop_db(db_name[, *link_id])*	Drops specified MySQL database.
mysql_errno([link_id])	Returns ID of error.
mysql_error([link_id])	Returns text error message.
mysql_fetch_array(result_id[,*result_type])*	Fetches result set as associative array. Result type can be MYSQL_ASSOC, MYSQL_NUM, or MYSQL_BOTH (default).
mysql_fetch_field(result_id[,*field_offset])*	Returns information about a field as an object.
mysql_fetch_lengths(result_id)	Returns length of each field in a result set.
mysql_fetch_object(result_id[,*result_type])*	Fetches result set as an object. See mysql_fetch_array for result types.
mysql_fetch_row(result_id)	Fetches result set as an enumerated array.
mysql_field_name(result_id, *field_index)*	Returns name of enumerated field.
mysql_field_seek(result_id, *field_offset)*	Moves result pointer to specified field offset. Used with mysql_fetch_field.
mysql_field_table(result_id, field_offset)	Returns name of specified field's table.
mysql_field_type(result_id, field_offset)	Returns type of offset field (for example, TINYINT, BLOB, VARCHAR).
mysql_field_flags(result_id, field_offset)	Returns flags associated with enumerated field (for example, NOT NULL, AUTO_ INCREMENT, BINARY).
mysql_field_len(result_id, field_offset)	Returns length of enumerated field.
mysql_free_result(result_id)	Frees memory used by result set
mysql_insert_id([link_id])	Returns AUTO_INCREMENTED ID of INSERT; or FALSE if insert failed or last query was not an insert.
mysql_list_fields(database, table[, link_id])	Returns result ID for use in mysql_field functions, without performing an actual query.
mysql_list_dbs([link_id])	Returns result pointer of databases on mysqld. Used with mysql_tablename.
mysql_list_tables(database[, link_id])	Returns result pointer of tables in database. Used with mysql_tablename.
mysql_num_fields(result_id)	Returns number of fields in a result set.
mysql_num_rows(result_id)	Returns number of rows in a result set.
mysql_pconnect([host][:port][:socket][, username][, password])	Opens persistent connection to database. All arguments are optional. Be careful — mysql_close and script termination will not close the connection.
mysql_query(query_string[, link_id])	Sends query to database. Remember to put the semicolon outside the doublequoted

	query string.
mysql_result(result_id, row_id, field_ identifier)	Returns single-field result. Field identifier can be field offset (0), field name (FirstName) or table-dot name (myfield.mytable).
mysql_select_db(database[, link_id])	Selects database for queries
mysql_tablename(result_id, table_id)	Used with any of the mysql_list functions to return the value referenced by a result pointer.