Accessing MySQL from PHP
by George Yohng

This guide is an attempt to provide more or less complete information on accessing MySQL API functionality from PHP web scripting language.

Today PHP is one of the most usable web languages used for server-side scripting. The usability of PHP came obvious since version 3 was released, and version 4 introduced another amount of useful features. A number of companies already moved from Perl/CGI to PHP, and more still to come.

PHP engine is typically used with Apache server under POSIX-compatible operating systems (Linux, FreeBSD, Solaris, AIX, other UNIX clones, etc.). However PHP itself isn't dependent on a particular web-server or operating system. As information changes enough frequently, my recommendation would be to visit official PHP site for a complete list of features, operating system supported and other information on PHP:

`http://www.php.net`

In the whole guide, PHP 4.0.6 or later version is assumed, however most of information will also be applicable to earlier versions of PHP. All OS-specific stuff will be marked as such.

PHP functionality, including installation process, is about the same for all POSIX-compatible operating systems, while it slightly differs for Microsoft Windows, and thus two MySQL-API installation sections are included to this guide.

The information provided in this guide should be mostly OS-independent. All example scripts were tested on SuSE Linux 7.2 Professional with Apache and PHP4 packages installed with default options, and Microsoft Windows 2000 with Apache web-server version 1.3.14 and Win32-precompiled PHP version 4.0.6.

PHP Basics

PHP – is a scripting language for writing web applications that execute on server-side. Scripting language itself is much alike C, however it still contains many differences. Here are some basic ones:

- While C program needs to be compiled before execution, PHP script is interpreted at runtime;
- Variables do not need to be declared in PHP;
- PHP operates more gently with string variables (e.g. "5" + "6" = 11);
- C is function-oriented language, while in PHP for execution code can be written straightforward;
- All variables in PHP should have "$" prefix, otherwise identifiers are treated as string constants;
- PHP doesn't make use of standard header files and libraries – all functions are built-in, file inclusion is mostly used for user extensions and text blocks;
- PHP supports hash arrays as native type;
- PHP code should be embedded to some different file (like MS-ASP scripting, or JavaScript) and is extracted by PHP engine before execution, while C requires a complete source file;
- PHP should not execute any user query or delayed request functions; PHP normally cannot display application windows, buttons or other widgets. Its purpose is to generate content (e.g., generation of text files, images, data files, etc);

Loop statements, conditionals and comments are identical to C syntax, with exception that each variable should be prepended with dollar sign "$". PHP outputs data with "echo" command.

The essential PHP concept is to embed scripting code to HTML, using unique delimiting tags to separate raw HTML from script code. PHP scripts are executed on server side (like Perl/CGI or Java servlets), and the final user gets pure HTML in a browser.

Provided that PHP code is surrounded with *<?php … ?>* tags, web server is able to filter such sections and puts the execution result instead.

Here's an example of sample PHP script, embedded to HTML:

```
<html>

<?php

    // Calculate FOO variable
    $foo = 2 + 2;

    // Output sample string to browser
    echo "PHP says Hello!<br><br>";

    // Output string with embedded value
```

```
    echo "2 + 2 = $foo";

?>

</html>
```

And the upper example will output this:

```
<html>

PHP says Hello!<br><br>2 + 2 = 4

</html>
```

This sample code doesn't include any calls to MySQL API; it simply shows the basics of outputting data from PHP script.

Considering code from the upper example, *$foo* – is a calculated variable (PHP requires dollar sign '$' to be put before each variable name). If *$foo* value was fetched from database instead, the same technique is used to output its content to the browser.

The following example shows trivial connection to database server and fetching data from there. In further examples HTML code will be mostly omitted, and only PHP code (fully or partially) will be shown, like this:

```
<?php
    // Connect to database server
    $hd = mysql_connect("myhost", "username", "password")
          or die ("Unable to connect");

    // Select database
    mysql_select_db ("database", $hd)
          or die ("Unable to select database");

    // Execute sample query
    $res = mysql_query("SELECT * FROM customer", $hd)
          or die ("Unable to run query");

    // Query number of rows in rowset
    $nrows = mysql_num_rows($res);

    // Output
    echo "The query returned $nrows row(s):\n\n";

    // Iteration loop, for each row in rowset
    while ($row = mysql_fetch_assoc($res))
    {
        // Assigning variables from cell values
        $data1 = $row["title"];
        $data2 = $row["fname"];
        $data3 = $row["lname"];
        $data4 = $row["phone"];
```

```
        // Outputting data to browser
        echo "ROW# $nr : $data1 $data2 $data3 $data4\n";
    }

    mysql_close($hd);
?>
```

Adding MySQL Support to PHP

This section will describe how to check and enable MySQL API support for PHP. If you are 100% sure that PHP is properly installed on your web server with MySQL support, you can safely skip all installation instructions.

Checking MySQL support availability

Before trying to run any MySQL-dependent PHP script, it is necessary to ensure that PHP on web server has such API installed.

Single line PHP sample below will query all the information about currently installed PHP modules and web server environment variables, and show them up in user-friendly form:

```
<?php phpinfo() ?>
```

Save this single line to *phpinfo.php* file within web server space, and load it through your browser. Note, that simply loading this file from local disk will not work – you'll have to pipe it through web server. In other words, you should make it work by loading some URL, which starts with "http://". The sample URL may look like:

```
http://localhost/phpinfo.php
```

If using correct URL you've got an empty page, or the page shows up unmodified example code, PHP is not installed to your web server, and PHP module requires installation itself.

If the information page appeared correctly, seek for "mysql" on the page. There should be a separate section called "mysql" and it will look like this:

mysql

MySQL Support	enabled
Active Persistent Links	0
Active Links	0
Client API version	3.23.32

If such looking part is shown on the information page, then mysql module is enabled and is working correctly, so you can skip "Enabling…" sections of this guide.

If the information page appeared, but no "mysql" section found, then MySQL API support needs to be installed to PHP and enabled.

Enabling MySQL API support for POSIX-compatible OS

When using POSIX-compatible OS (e.g., Linux, FreeBSD, Solaris, etc), then MySQL-enabling option should be applied while compiling PHP from distribution.

> Note, that if precompiled PHP4 is included to your OS installation package, MySQL support is possibly included there by default.

If you've got to compile PHP yourself, specify *--with-mysql* option to configure script. After proper compiling and installation, MySQL support should be enabled. You can check it again by using *phpinfo* function sample (above in this guide).

Enabling MySQL API support for Microsoft Windows

When running Microsoft Windows, you may run into several issues with PHP itself and MySQL. While C compiler is an essential part of most POSIX-compatible operating systems, Windows system typically doesn't contain any compiler installed. Anyway, compiling PHP under Windows is pain, if you never did this before; so consider downloading Win32-precompiled version from PHP site.

The Win32-precompiled version is available for download from official PHP site, and contains MySQL support integrated.

Using MySQL API in PHP

PHP provides a set of functions to use for accessing and manipulating data on MySQL server. The following sections will provide a step-by-step description of how to create and manage MySQL connections, work with MySQL tables, insert and remove data, etc.

Database Connections

Before any operations are to be made on the data, database connection should be established. Two general types of connection exist – normal connections and persistent connections.

There is no commonly used term for mentioning normal (non-persistent) connections. Thus, when connection is mentioned to be "normal", or simply not mentioned to be persistent, then it is likely non-persistent connection.

Within simple PHP scripts, single-time MySQL connections are created, and closed once script execution is completed. This is good for rare connections, when PHP page isn't requested too frequently.

The basic idea of persistent connections is to keep connection open for some particular time, and if the page loads multiple times, PHP code will reclaim the same connection.

For high traffic web sites, it would be reasonable to use persistent connections. However, if web site suffers from a very high traffic (million visits per day), for used PHP and MySQL versions the practice showed that performance degrades.

For such web sites, I would not recommend to use persistent connections. PHP developers reported, that under really heavy load, persistent connections reclaim too much of web server resources, and thus performance degrades, as the result of continuous memory swapping. However, if you are not planning to run such heavy traffic sites (e.g., like world immigration center, or Microsoft), persistent connection will act with noticeable performance boost, comparing to normal non-persistent connections.

As for scripting, persistent connection doesn't differ from usual connection, except that different function is used for connection establishment. Trying to close persistent connection will do no effect, however it's often useful to keep closing function calls in PHP code (e.g., for compatibility purposes).

Establishing connection

To establish simple connection, function *mysql_connect* should be used.

```
$handle = mysql_connect ( host [, username[, password ]] )
```

Working with persistent connection differs with only one feature – connection function name is *mysql_pconnect*. The function has the same parameter meanings and is used exactly in same way.

```
$handle = mysql_pconnect     ( host [, username[, password ]] )
```

All three parameters are of type string, and connection handle is returned once function is executed. For example, if connection to server *"cassy"* is desired, the line from PHP script may look like this:

```
$link = mysql_connect ("cassy", "george", "greatpassword1105");
```

In the upper example, user name is *"george"* and the password is *"greatpassword1105"*.

If name and password are not specified, the PHP process owner's user ID is taken, and empty password is assumed. If host name is not specified, *"localhost"* will apply.

You will not be able to specify user name, if *sql.safe_mode* option is set in *php.ini* file. In this case, default user will be used instead.

The connection line can also specify port number to connect to. By default, port 3306 is assumed. However if MySQL server is installed on different port, specifying it explicitly will help:

```
$link = mysql_connect ("cassy:4444", "george",
"greatpassword1105");
```

Also, it's possible to connect local server using named sockets:

```
$link = mysql_connect (":/tmp/mysql.sock", "george",
"greatpassword1105");
```

If the connection should perform on localhost with default port number, first parameter can be omitted by specifying *null* instead:

```
$link = mysql_connect ( null, "george", "greatpassword1105");
```

In this example, *null* is not a variable, but a keyword, thus no dollar sign '$' is put before.

How to check this against errors? Normally, if server data is specified properly and MySQL is accessible, such connection can always be established, however sometimes (e.g., due to server name misspelled, wrong configuration, heavy load, flaw or software bug) such connection will fail. If this happens, in most cases further PHP script execution is useless. You can terminate PHP script with diagnostic message by using *die* function, like this:

```
$link = mysql_connect ( null, "george", "greatpassword1105");

if (!$link) die("Can't connect to database server");
```

Using some essential PHP features, the upper code can be optimized to look like this:

```
$link = mysql_connect ( null, "george", "greatpassword1105")
          or die("Can't connect to database server");
```

More information on error handling could be found under corresponding title later in this guide.

Under some circumstances, connection to "localhost" will fail with this string in output:

```
Can't connect to local MySQL server through socket
'/tmp/mysql.sock' (2)
```

In this case, first parameter should explicitly specify 127.0.0.1 as host name (instead of "localhost" or *null*). You may also want to add port number.

Not depending on whether die function is used or not, MySQL will output diagnostic messages (warnings) itself. To suppress such warnings, put @ sign in front of function name. This is typically to be done on a completed and tested PHP application, as keeping MySQL warnings is often helpful for debugging purposes.

After a successful connection, *$link* variable will contain connection handle. This handle is to be used further in all MySQL API function calls.

> Note, that passing password as raw strings (like in the examples above) is suitable while PHP code is not accessible in the source form from the network. Otherwise, password should be stored somewhere else, possibly in private directory.

Selecting database

After connection is established, the default database should be selected to use for SQL queries, which don't specify database name explicitly. The function *mysql_select_db* is used for such purposes.

```
mysql_select_db( database_name [, connection_handle] )
```

The very first parameter specifies name of database to select within connected server space; *connection_handle* – is the variable, resulted earlier from *mysql_connect* function. PHP manual describes, that *connection_handle* can be omitted, if last opened handle is to be used, however this can lead to confusion, when multiple connections are used in the script. So I recommend to specify *connection_handle* explicitly, when more than one connection is planned (e.g., if you are writing some kind of abstraction layer or PHP database engine).

Function *mysql_select_db* returns either TRUE on successful database selection, or FALSE on error (e.g., database not found, connection handle is bad, lightning hit to the server, etc).

The example below connects MySQL server, selects database, executes sample query (to delete all data from a particular table) and closes the connection:

```
<?php
    // Connect to database server
    $hd = mysql_connect("localhost", "george",
                                     "georgespassword9798")

          or die ("Unable to connect");

    // Select database
    mysql_select_db ("andrewsbase", $hd)
          or die ("Unable to select database");

    // Execute sample query (delete all data in customer table)
    $res = mysql_query("DELETE FROM customer",$hd)
          or die ("Unable to run query");
```

```
    // Close connection
    mysql_close($hd);

?>
```

Sometimes it is desired to use multiple queries on multiple databases in the same PHP script. Of course, it's pretty valid to use *mysql_select_db* function before each query, which requires different database to operate on. But if PHP script mostly works with one database, and database switches are made only for few queries, specifying database name explicitly is the best way. The upper example needs not to select current database, if such query is used:

```
...
    // Execute sample query (delete all data in customer table)
    $res = mysql_query("DELETE FROM andrewsbase.customer",$hd)
          or die ("Unable to run query");
...
```

Closing connection

Finally, after connection is not needed anymore, *mysql_close* call should be used to close connection, as you probably guessed yourself from the example in previous section.

```
mysql_close ( connection_handle )
```

Nothing wrong happens when trying to close persistent connections – the function will simply perform no operation, however to preserve compatibility, even for persistent connections I would recommend keeping *mysql_close* call anyway (what if in the future it would be desired to change to simple connections?).

PHP documentation mentions, that it is not necessary to use *mysql_close* at all, however the practice showed, that opposite to manual, unclosed orphan connections are kept in memory, and are closed only in some time (after timeout expires). This is, of course, a resource black hole for web sites under heavy load.

Obtaining information by connection handle

Starting with PHP version 4.0.5, few *mysql_get_xxxx_info* functions were introduced. These functions will help to obtain some basic information about MySQL API and connection handles:

```
$text    = mysql_get_client_info  ( )

$text    = mysql_get_server_info  ( connection_handle )

$text    = mysql_get_host_info    ( connection_handle )

$val     = mysql_get_proto_info   ( connection_handle )
```

Function *mysql_get_client_info* will return a string, containing current PHP-MySQL client library version. Note, that this version number is not related to MySQL version installed on server.

Either internal PHP-MySQL API library, or MySQL-provided library can be used during PHP compilation. If path to MySQL libraries is not specified during PHP compilation, PHP uses built-in MySQL client library, which typically is older than it could be otherwise. The value of *mysql_get_client_info* will look like this:

```
3.23.32
```

Function *mysql_get_server_info* needs *connection_handle* parameter; it queries MySQL server version, using the server connected via *connection_handle*. The return value will be formatted exactly in the same way, as in *mysql_get_client_info* function, however these two values are not related to each other, and *mysql_get_server_info* will typically return different value. For example, on my machine it indicates:

```
3.23.37
```

Functions *mysql_get_host_info* and *mysql_get_proto_info* are used to get more information on currently connected host and protocol.

The example below demonstrates usage of *mysql_get_xxxx_info* functions:

```
<html><pre>
<?

    $hd = mysql_connect("192.168.1.2","root","")
            or die("Can not connect");

    echo "mysql_get_client_info:  " . mysql_get_client_info() . "\n";
    echo "mysql_get_server_info:  " . mysql_get_server_info($hd) . "\n\n";

    echo "mysql_get_host_info:    " . mysql_get_host_info($hd) . "\n";
    echo "mysql_get_proto_info:   " . mysql_get_proto_info($hd) . "\n";

    // Close connection
    mysql_close($hd);

?>
</pre></html>
```

And this outputs the following data on my machine:

```
mysql_get_client_info:  3.23.32
mysql_get_server_info:  3.23.37

mysql_get_host_info:    192.168.1.2 via TCP/IP
mysql_get_proto_info:   10
```

Executing queries

To fetch or alter data, SQL queries are to be executed. "SELECT" queries will obviously have result, and the description of how to deal with rowsets is provided later in this guide. Less obvious is, that "DELETE", "INSERT" and "UPDATE" queries will have result as well, however the result will contain no rowset, but different miscellaneous information, such as number of rows affected, etc. This section will explain how to execute queries and how to reclaim those "unobvious" results.

Executing raw SQL

MySQL API for PHP contains function named *mysql_query*, which takes query string and connection handle as parameters. This function either returns rowset handle, miscellaneous info, or zero value (in case of errors). PHP itself doesn't differ rowset handles from miscellaneous info handles, and in terms of PHP returned value is called "resource handle"; however, different functions are used to fetch data from rowsets, than to get number of affected rows from miscellaneous handle, etc.

```
$res = mysql_query( SQL_query_string [, connection_handle] )
```

If *connection_handle* parameter is omitted, then the last connection is used. If you plan to run queries on multiple connections (e.g., copying data from one database server to another, or synchronizing two database servers), omitting *connection_handle* parameter can lead to confusion, so if you plan to handle two connections in the same PHP script, specify *connection_handle* parameters explicitly.

The *SQL_query_string* parameter specifies SQL query string to execute. Note, that no semicolon should be put to the end of string.

To execute sample SELECT query, you can put it this way:

```
$res = mysql_query( "SELECT * FROM customer", $hd );
```

In the upper example, *$hd* is a valid connection handle. And *$res* – is the target variable for receiving rowset handle. In case of DELETE, INSERT or UPDATE statements usage, *$res* receives miscellaneous information handle. To show how to work with such handles, a slightly modified table clearing example is provided below:

```
<?php
    // Connect to database server
    $hd = mysql_connect("localhost", "george",
                                    "georgespassword9798")
          or die ("Unable to connect");

    // Execute sample query (delete all data in customer table)
    $res = mysql_query("DELETE FROM andrewsbase.customer",$hd)
          or die ("Unable to run query");

    // Display number of rows affected
```

```
    echo "Rows deleted: " . mysql_affected_rows($res);

    // Close connection
    mysql_close($hd);

?>
```

MySQL API for PHP contains function *mysql_affected_rows*, used to obtain number of rows affected by the query operation. Note, that this one doesn't work for SELECT statement queries – it works for data-modifying queries only; there's a different function to obtain number of rows attached to rowset handles (described later in this guide).

```
$nrows = mysql_affected_rows ( result_handle )
```

This function receives *result_handle*, returned by *mysql_query* function; *$nrows* – is an integer variable assigned to affected rows count.

Note, that when used with UPDATE statement queries, *mysql_affected_rows* can actually return different number of rows – those records, which already contained desired values would not be counted.

Formatting data for queries

In previous section, we went through description for raw SQL query execution. This would be enough, when all data for query is constant, pre-formatted and already prepared for execution. However, dynamically generated query strings may cause problems, if formatted incorrectly.

Suppose, we need to execute this query, where *$pm* is a parameter variable filled before:

```
$res=mysql_query("SELECT * FROM customer WHERE lname='$pm' ");
```

When $pm is equal to "Smith", the query expands to the following:

```
SELECT * FROM customer WHERE lname='Smith'
```

This will execute fine. Nothing wrong happens, when this query is executed on these customers: "Jenny Stones", "Simon Cozens", "Dave Johns"… Finally, the list ends with "Bill O'Neill", and PHP application stops working as expected. The query expands to:

```
SELECT * FROM customer WHERE lname='O'Neill'
```

In the upper case, the query will simply fail with "O'Neill". But what if someone was accidentally (by parents) called "George O';DROP DATABASE andrewsbase" ? While being fully legal, the query will expand to:

```
SELECT * FROM customer WHERE lname='O';
```

```
DROP DATABASE andrewsbase
```

For this query, the last thing you want is to drop database. This is typical example of possible security holes – such security holes may exist in any type of PHP application using SQL queries. If such security issue is once missed on some web site, potentially any data from MySQL server can be destroyed.

Actually PHP doesn't support multiple queries in one line, but anyway, while not blocking this issue, MySQL query may be expanded to return listing of some secret or private data, or to show up multiple names in a row.

Obviously, to fix this query the apostrophe ' should be prefixed with a backslash. There are other special symbols, which should be prefixed as well to avoid query confusion. Function *mysql_escape_string* takes care of all such cases and prepares string for inserting to queries:

```
$text = mysql_escape_string( unprepared_string )
```

Here *$text* means any available string variable, and *unprepared_string* – is a parameter, which contains original value (e.g. user name).

So the upper example should be modified to make all alien names fetch to query correctly:

```
// Escape $pm string
$pm = mysql_escape_string ( $pm );

// Execute SELECT query
$res=mysql_query ("SELECT * FROM customer WHERE lname='$pm' ");
```

When "O'Neill" will apply, the query will expand to:

```
SELECT * FROM customer WHERE lname='O\'Neill'
```

MySQL will understand this correctly, and will execute correct query.

> Note, that in some special cases you may want to escape % and _ characters as well, but *mysql_escape_string* function won't do this for you. However, normally these characters don't have to be escaped.

Similar technique can be used to prepare data for inserting and updating records.

Working with rowsets

It was mentioned earlier in this guide, that SELECT statement query returns some special kind of result – rowset handle. In this section, I will describe the common ways

to pull actual data from rowset handles and how to deal with such rowset handles in some other aspects.

Buffered queries

SELECT statement queries are executed like all other queries – with *mysql_query* function:

```
$res = mysql_query("SELECT xxxx FROM yyyy WHERE zzzz",
$connection_handle);
```

After such command execution, *$res* variable will contain rowset handle, to which buffered rowset is attached. By function *mysql_query*, the complete rowset is received from SQL server and stored in memory.

Normally, rows are extracted in sequential order, as further rows may not be ready, like in case of unbuffered queries (described in next section). Function *mysql_query* fully buffers received data, so rows can be accessed in random order. You can seek to some particular row by using *mysql_data_seek* function:

```
mysql_data_seek ( rowset_handle , row_number )
```

The first parameter is actually the result (e.g., *$res* variable) returned by *mysql_query* function. The second parameter means the destination row number, starting with 0.

Function returns logical value. *TRUE* will be returned, if row was successfully found and positioned, *FALSE* otherwise. All *mysql_fetch_xxxx* functions will start fetching from positioned row, or by default (if row pointer wasn't changed) from the beginning of rowset.

Total number of rows in rowset can be obtained by *mysql_num_rows* function:

```
$nrows = mysql_num_rows ( rowset_handle )
```

As *mysql_query* buffers all data received from MySQL, total number of rows can be queried simply by specifying *rowset_handle*, returned from earlier call to *mysql_query*.

Function *mysql_num_fields* queries number of fields per row:

```
$nfields = mysql_num_fields ( rowset_handle )
```

Function *mysql_num_fields* can be useful while retrieving information about table columns.

Those two functions query the dimensions of rowset, so by multiplying *$nfields* by *$nrows*, we get the total number of fields in rowset. This can be used in data size estimation for tables of equally-typed columns.

Unbuffered queries

While *mysql_query* waits for the query to complete and buffers the result, *mysql_unbuffered_query* function returns as soon as query is passed to SQL server.

Sometimes, database size does not allow storing all queried results in memory (e.g., database is about 14GB), thus in these cases, for performance issues it's preferable to use *mysql _unbuffered_query* function instead of *mysql_query*.

```
$res = mysql_unbuffered_query ( query_string [,
                                connection_handle ] )
```

Unbuffered queries are especially useful when not each queried value is needed (like search engines).

However, unbuffered query results cannot be scanned for number of rows, cannot be sought randomly, cannot be used for multiple queries, etc.

> Unbuffered query data results are lost once sequential query on the same connection is executed.

```
...
    // Get result from querying table customer
    $res1=mysql_unbuffered_query("SELECT * FROM customer",$hd);

    // Get result from querying table item (ATTENTION)
    $res2 = mysql_unbuffered_query("SELECT * FROM item", $hd);

    // * AT THIS POINT, $res1 is no more accessible due to
    // * all unbuffered data overwritten by seqiential query
...
```

Fetching rows from rowsets

The essential way to extract data row, is to use *mysql_fetch_row* function:

```
$array = mysql_fetch_row( rowset_handle )
```

After such execution, $array will contain data row, which can be accessed by numeric index, starting with 0. E.g., if table contains eight fields: *customer_id, title, fname, lname, addressline, town, zipcode, phone* exactly in this order, *$array[0]* is assigned to value of field *customer_id*, *$array[1]* correspondingly to *fname* value, and so on.

If fields order is fixed, the following code can be used for assigning field values to variables in one PHP line:

```
list($idv, $title, $fname, $lname,
```

```
    $address, $town, $zip, $phone) = mysql_fetch_row($res);
```

Once no row can be fetched (e.g., all rows were fetched before, or query returned no data), zero value is returned instead of array.

The easiest way to output all data from table, is to use the following code:

```
...
    // Execute SELECT statement query on connection,
    // associated with $hd

    $res = mysql_query("SELECT * FROM customer", $hd)
        or die ("Query execution failed");

    while(1)
    {
        // Fetch one row
        $row = mysql_fetch_row($res);

        // If no data row found, exit loop
        if (!$row) break;

        // Perform loop on each value in array
        for($nf=0; $nf<count($row); $nf++)
        {
            // Assign value
            $val = $row[$nf];

            // Output single field sequential number and data
            echo "Field # $nf = $val  \n";
        }

        // Separate rows from each other
        echo "----- \n";
    }
...
```

The upper example can be optimized to this:

```
...
    // Execute SELECT statement query on connection,
    // associated with $hd
    $res = mysql_query("SELECT * FROM customer", $hd)
        or die ("Query execution failed");

    // Run loop until data exceeds
    while($row = mysql_fetch_row($res))
    {
        // For each value in array...
        foreach($row as $nf => $val)
        {
            // Output single field sequential number and data
            echo "Field # $nf = $val  \n";
```

```
        }

        // Separate rows from each other
        echo "----- \n";

    }
...
```

Obviously, this isn't the most flexible way to deal with data. By this, you can know field values only. Different PHP functions allow retrieving associative hash with field names instead of indexed array:

```
$hash = mysql_fetch_assoc( rowset_handle )
```

In this case, fields values can be referred as *$hash["customer_id"], $hash["fname"]*, etc. The last "optimized" example above will also work for hashes. In this case you will get output like this:

```
Field # customer_id = 1
Field # title = Miss
Field # fname = Jenny
Field # lname = Stones
Field # addressline = 27 Rowan Avenue
Field # town = Hightown
Field # zipcode = NT2 1AQ
Field # phone = 023 9876
-----
Field # customer_id = 2
Field # title = Mr
Field # fname = Andrew
Field # lname = Stones
Field # addressline = 52 The Willows
Field # town = Lowtown
Field # zipcode = LT5 7RA
Field # phone = 876 3527
-----

...
...
...

-----
Field # customer_id = 15
Field # title = Mr
Field # fname = David
Field # lname = Hudson
Field # addressline = 4  The Square
Field # town = Milltown
Field # zipcode = MT2 6RT
Field # phone = 961 4526
-----
```

If you need to use field values frequently, obviously, it's not convenient to write *$row["FIELDNAME"]* each time, plus you can't easily embed such construction to string constant.

PHP provides function *extract* to create variables, which names are equal to hash keys.

```
...
    // Opening scope to prevent variable distribution
    // to other parts of script
    {
        // Get sequential row hash from rowset
        $rowhash = mysql_fetch_assoc($res);

        // Extract row hash to upper level
        extract($rowhash);

        // Now fields can be used directly
        echo "Datas: # $customer_id $fname $lname - $phone  \n"
    }
    // Scope closing bracket restricts extracted variable
visibility
...
```

The upper solution can be used only if you don't need to use some variables with such names, but not related to fields. E.g., if for the upper example table contains field named *"rowhash"*, I would not recommend using the upper solution, as it may overwrite *$rowhash* variable and cause unwanted side effects.

So if you aren't sure about correctness of value extracting, but still eager to embed field values to string constants, it can be done like this:

```
echo "Here's a value of FNAME – {$rowhash['fname']} ";
```

Notice the single quotes used instead of double quotes surrounding field name.

In addition, PHP contains *mysql_fetch_array* function:

```
$foo = mysql_fetch_array( rowset_handle [, fetch_type ] )
```

Depending on *fetch_type* parameter, function can work like *mysql_fetch_row*, *mysql_fetch_assoc*, or result a merged hash as if those two were called sequentially and their results merged together; *fetch_type* parameter can have the following values:

MYSQL_ASSOC ..... the function will act like mysql_fetch_assoc
MYSQL_NUM ..... the function will act like mysql_fetch_row
MYSQL_BOTH ..... will work like both functions merged

This function isn't much useful, since it may introduce some confusion.

First, as you probably guessed yourself, when specifying *MYSQL_BOTH* as *fetch_type*, function will return each value twice in hash – one with numeric key and another with named key.

> Second, *mysql_fetch_array* function will NOT add any numeric entries to hash, if the corresponding field values are set to NULL. So the actual number of values in hash may differ from fields count in table. PHP developers reported, that when a table record contains all fields set to NULL, using *mysql_fetch_array* might lead to even further confusion.

Finally, to fetch row to object variable, *mysql_fetch_object* function can be used:

```
$foo = mysql_fetch_object( rowset_handle [, fetch_type ] )
```

The second *"fetch_type"* parameter has the same meaning, as in function *mysql_fetch_array*. After row is fetched, object properties are assigned to corresponding field values.

If you specify *MYSQL_ASSOC* or *MYSQL_BOTH* as *fetch_type*, then later you can refer the returned field values as object properties (e.g., *$foo→fname*, *$foo→phone*, etc).

Querying information about columns in table

In some cases, it's necessary to execute queries on unknown tables (e.g., user-entered queries). For example, storing property values of complex structures sometimes will require different tables to be used with the same code. Thus it would be necessary to query information about table columns before actual data fetching. Earlier in this guide I described how to get number of rows in rowset and how to seek pointer to a particular row. Different PHP function is used to operate with column pointer:

```
mysql_field_seek ( rowset_handle , column_offset )
```

The second parameter *column_offset* specifies default offset for column pointer. Field info fetching function can explicitly specify different field offset, but if not specified, the default pointer value will be used.

To fetch information of sequential column in a rowset, function *mysql_fetch_field* is used.

```
$object = mysql_fetch_field( rowset_handle [, field_offset ] )
```

The object value is returned. You can obtain the following object properties:

$object→name	the column/field name
$object→table	name of table, to which rowset belongs
$object→max_length	maximum value length
$object→def	default field value
$object→not_null	1 if column is forced to be not NULL, 0 otherwise
$object→primary_key	1 if column is primary key for the table, 0 otherwise

$object→unique_key ..... 1 if column is unique key for the table, 0 otherwise
$object→multiple_key ..... 1 if column is non-unique key, 0 otherwise
$object→numeric ..... 1 if field is numeric, 0 otherwise
$object→blob ..... 1 if field type is BLOB, 0 otherwise
$object→type ..... the type of column/field
$object→unsigned ..... 1 if value stored in unsigned format, 0 otherwise
$object→zerofill ..... 1 if column is zero-filled, 0 otherwise

Of course, *mysql_fetch_field* function can't get any particular field value – it operates on column basis, not referring any particular row. This function affects neither row pointer, nor rowset data, but only gets basic information about table used in SELECT query, not depending on its contents.

For convenience, there are some PHP functions to obtain single property of a field:

```
$tablename   = mysql_field_table ( rowset_handle, field_offset)

$typestring  = mysql_field_type ( rowset_handle, field_offset)

$fieldlength = mysql_field_len ( rowset_handle, field_offset)

$sqlflags    = mysql_field_flags ( rowset_handle, field_offset)
```

All four functions receive *rowset_handle* and *field_offset* in parameters.

Note, that purpose of passing *field_offset* to *mysql_field_table* function is not obvious. Normally, all fields in a row set belong to the same table, however for some complex merged rowsets it's possible that single row will contain fields, which belong to different tables. Of course this is up to a particular PHP application, and if you don't ever plan to use merged rowsets, you can safely pass 0 as the second parameter for *mysql_field_table*.

The following example shows the easy way to output table structure:

```
<html><pre>
<?php

    // Connect to database server
    $hd = mysql_connect("andrewshost", "george",
                        "whocaresaboutsuchlongpasswords")
          or die ("Unable to connect");

    // Execute query
    $res=mysql_query("SELECT * FROM andrewsbase.customer", $hd)
           or die("Unable to execute query");

    // Get dimensions of table customer
    $nfields = mysql_num_fields ($res);
    $nrows   = mysql_num_rows ($res);

    // Get table name (for this example, it will
```

```
    // be equal to "customer",
    // but will change once the upper query changed)
    $table = mysql_field_table ($res, 0);

    // Output header
    echo "Table $table has ".
         "$nfields field(s) and $nrows record(s).\n";

    echo "Table structure:\n";

    // For each field in table...
    for($t=0; $t<$nfields; $t++)
    {
        // Query information
        $type  = mysql_field_type  ($res, $t);
        $name  = mysql_field_name  ($res, $t);
        $len   = mysql_field_len   ($res, $t);
        $flags = mysql_field_flags ($res, $t);

        // Output line
        echo "$type $name $len $flags\n";
    }

    // Close connection
    mysql_close($hd);
?>
</pre></html>
```

This is sample output for the upper example:

```
Table customer has 8 field(s) and 15 record(s).
Table structure:
int customer_id 11 not_null primary_key auto_increment
string title 4
string fname 32
string lname 32 not_null
string addressline 64
string town 32
string zipcode 10 not_null
string phone 16
```

Freeing rowsets

All rowsets are automatically freed upon script execution completes. However, if PHP script is intended to receive some huge data arrays, and when especially doing this in cycle, memory space will be taken and not returned back until script completion.

Normally, if script contains only one call to query, freeing rowset is not necessary, however this should be done, if query resulting rowset is not necessary anymore before another query is about to start:

```
mysql_free_result ( rowset_handle )
```

Notice, that first parameter is a rowset handle, not result handle. This function should be used to free rowsets, obtained by SELECT statement queries, and should NOT be used for other types of query.

This is a typical example of code, where using *mysql_free_result* function will help, however this will need customer ZIP codes in base changed to values from 1 to 15:

```
...
    // Running loop for $grp from 1 to 15
    for($grp=1; $grp<=15; $grp++)
    {
        // Sample output...
        echo "Outputting data for ZIP # $grp: \n";

        // Executing query on some database
        $res=mysql_query(
             "SELECT * FROM customer WHERE zipcode=$grp", $hd)

                or die("Unable to perform query on ".
                       "zipcode # $grp.");

        // Inner loop to fetch all records from $res
        while ($hash = mysql_fetch_assoc($res))
        {
            echo "Id:    {$hash['customer_id']} \n";
            echo "Name:  {$hash['fname']} \n";
            echo "Phone: {$hash['phone']} \n\n";
        }

        // Freeing rowset before next iteration
        mysql_free_result($res);

        // Delimiter between output of adjacent queries
        echo "----- \n";
    }
...
```

Note, that in most cases you won't notice the difference, however if huge rowsets are returned, freeing rowsets will help.

Type conversion of data values

Because PHP always returns MySQL field values as strings, you may run into several troubles when using operations over retrieved variables. For example, bitwise operations will work not as you may expect, if two operand numbers are retrieved from MySQL database.

If you are about to do some arithmetic or other type-dependent operation on fetched values, use *settype()* function to specify type explicitly before. For example, if $row is fetched from rowset, to specify type for field ID execute:

```
settype($row["customer_id”], "integer");
```

Function *settype* can receive the following constants as its second parameter:

```
"integer"
"double"
"string"
"array"
"object"
```

Error Handling

PHP provides two general functions for getting status information on the last MySQL API function execution. You can either get error number with *mysql_errno* function, or the full message text with *mysql_error*.

```
$errNr = mysql_errno()

$errtext = mysql_error()
```

Function *mysql_errno()* provides numerical status code of the last executed MySQL operation. If operation was successful, then zero value is returned; otherwise function will return error code, generated by MySQL.

Function *mysql_error()* returns error message string. If actual meaning of error is not important for script functionality (e.g. script simply quits on any error occurrence), while you still want to show the error message up to user, *mysql_error* function will be useful.

The example below shows how to deal with *mysql_errno* and *mysql_error* functions:

```
<html><pre>
<?

    $hd = mysql_connect("192.168.1.2","root")
            or die ("Unable to connect");

    $res = mysql_query("SEEK FOR GRANDMA, PLEASE",$hd);

    echo "mysql_errno() = ".mysql_errno()."\n";
    echo "mysql_error() = ".mysql_error()."\n";

    mysql_close($hd);

?>
</pre>
</html>
```

Provided that connection was established successfully, the upper example will output:

```
mysql_errno() = 1064
mysql_error() = You have an error in your SQL syntax near 'SEEK
FOR GRANDMA, PLEASE' at line 1
```

PEAR

PHP community made an attempt to create open source code repository (like CPAN is for Perl). As the result, PEAR appeared.

PEAR stands for **P**HP **E**xtension and **A**pplication **R**epository, and represents a large collection of object oriented open source PHP classes.

For now, PEAR isn't documented well. Instead, some of information can be found on PHP official web site, and the other part – as comments in PEAR source code.

To use PEAR, you must have some knowledge of object oriented programming in relation to PHP. PHP object oriented programming techniques are described here:

```
http://www.php.net/manual/en/language.oop.php
```

You can find more information on PEAR structure and some good tutorials at the following links:

```
http://pear.php.net
http://php.weblogs.com/php_pear_tutorials
```

Next sections will briefly describe how to perform database connections using PEAR, how to execute queries, fetch results and handle erroneous situations.

Getting PEAR to work

Currently, PEAR is a part of PHP distribution. PEAR sources can be found under the directory *pear*, relative to PHP installation.

To enable PEAR functionality, PHP include path variable should be adjusted to include PEAR directory as well. For this, open *php.ini* file in any text editor and seek for *"include_path"* variable. Normally on fresh installations this variable will contain no value, so you just simply have to put full path to PEAR there. Otherwise, if some paths are already used, you can add PEAR's directory to the end of list. Place a colon to separate multiple path entries from each other.

In some cases you will need current directory to present in path (for including local files), so add `.' as additional path entry – this will allow inclusion of files from current script directory.

To enable PEAR DB functionality for PHP, you should put the following line to the very begging of script:

```
require_once "DB.php";
```

This will seek for *DB.php* in include path (which should include PEAR path as well), and include its contents immediately.

PEAR's database abstraction interfaces

PEAR's DB abstraction class allows easy manipulation with databases, not depending on server type. No matter whether it would be MySQL or PostgreSQL – to port code from one database to another generally less lines of code will be changed, as opposed to full database-manipulation code reimplementation, when using traditional approach.

PEAR interface also adds convenient features to work with multiple results and advanced error handling, by providing corresponding classes and objects. In basic concept, PEAR's DB abstraction idea is much similar to Perl's DBI module.

The code flexibility is gained at a cost of performance. Typically software developer's time is more expensive than machine time, however some software solutions may require better performance, than PEAR can provide. And if application is planned to use MySQL-server databases only, it could be reasonable to use standard MySQL API, which provides generally better performance and is typically easier to use in simple projects.

Using PEAR's DB interface

With a purpose to illustrate the basics of PEAR's DB interface, the example is provided below, which performs a connection to database server, executes query and pulls some data from row set. Conceptually the sample code is built like other samples in this guide, thus looking at the corresponding comments, it would be easy to get understanding of how to deal with PEAR's DB:

```
<html><pre>
<?php

    // Turn on PEAR's DB functionality
    require_once "DB.php";

    // Database connection parameters
    $user     = "george";
    $pwd      = "georgeslongpassword";
    $host     = "localhost";
    $database = "andrewsbase";

    // Format parameters into DSN
    $dsn = "mysql://$user:$pwd@$host/$database";

    // Perform a connection
    $hd = DB::connect($dsn);

    // Check, whether the connection succeeded
    if (DB::isError($hd))
    {
        // If an error occurred, show message
        // and terminate the script

        die ($hd->getMessage());
    }
```

```
    // Execute SELECT statement query
    $res = $hd->query("SELECT * FROM customer");

    // Check for query execution status
    if (DB::isError($res))
    {
        // If an error occurred, show message
        // and terminate the script

        die ($res->getMessage());
    }

    // Fetch rows in a cycle
    while ( $row = $res->fetchRow(DB_FETCHMODE_ASSOC) )
    {
        // Output data
        echo "Customer's first name is {$row['fname']}\n";
        echo "Customer's last name is  {$row['lname']}\n";

        // Separate records
        echo "-----\n";
    }

    // Close connection
    $hd->disconnect();

?>
</pre></html>
```

As you see in the upper code example, functionality is much similar to MySQL API, however no MySQL functions are used directly.

Notice, that connection is done using so-called DSN string, which conceptually is similar to URL line in your browser, however in this case DSN is used to connect MySQL server via database protocol (and, of course, this string will not work in your browser).

PEAR's error handling

PEAR includes advanced technique of error handling. By using PEAR's function *setErrorHandling*, you can change the behavior of PEAR's DB, when an error occurs:

```
$hd->setErrorHandling( handling_policy )
```

Here *$hd* means PEAR's connection handle, obtained by *DB:connect* function. Parameter *handling_policy* can take the following values:

PEAR_ERROR_RETURN return an error object and continue execution (default policy)
PEAR_ERROR_PRINT print error message and continue execution
PEAR_ERROR_DIE print error message and terminate PHP script

PEAR_ERROR_TRIGGER use PHP's *trigger_error* function for advanced internal error handling

PEAR_ERROR_CALLBACK perform callback function call to handle erroneous situation

More information on PEAR's error handling can be found at this URL:

http://php.net/manual/en/class.pear-error.php

Summary

In this guide, the complete information on PHP and MySQL interaction was provided. In the very beginning, PHP basics were described and a sample code for using PHP scripting within HTML.

Through the next few sections, we learned how to install and enable MySQL API support to PHP engine. This, of course, was the easiest part to understand, as the complete recommendations were given on installation support. No "installation from source code" section was provided, as it could take one more complete guide, however with given instructions it is practically impossible to install precompiled PHP in wrong way (unless someone tries to install older version).

Next section block provided basic information on MySQL connections, connection types and how to avoid common mistakes, typically made at this point. Persistent connections were reviewed and described. As from both programmer's and user's side, persistent connections and standard connections look mostly the same, the examples for standard *mysql_connect* function should also be suitable for usage with *mysql_pconnect* instead.

The next few sections covered information on executing buffered and unbuffered queries, fetching data from given rowsets and working with type conversion. This is probably the longest part, but the structure of sections is more alike an exact tutorial on driving through crowds of SELECT statements and other database stuff. The information supplied should prevent reader from doing common database security mistakes, such as storing passwords in code, executing non-escaped query strings and naming children as *"George O;DROP DATABASE andrewsbase"*.

The "Error Handling" section supplied basics of error handling and reporting.

Sequential guide part was dedicated to PEAR Database Abstraction interface. While PEAR is not widely used yet, some users may find it to be useful for portable database projects, such as web engines, etc.

Finally, this chapter ended with "Summary" part, which described in briefly, that in this chapter, the complete information… …this is possibly what you have read already.