Writing efficient PHP

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. About this tutorial

Should I take this tutorial?

This tutorial targets the PHP developer who already understands PHP, but wants to write more efficient PHP code or to improve the performance of existing PHP applications. It is not intended as an introduction to PHP -- plenty of other resources are available for that. Rather, it assumes that you already have an installation of PHP available and are familiar with the basic PHP syntax.

In this tutorial, you will learn ways to improve the performance of your PHP code. The tutorial is in four main parts:

- General coding considerations that can be often applied to any language, but are illustrated with specific PHP examples (see Writing efficient code on page 4).
- Nuances of the PHP language and how you can use them to further improve the efficiency of your PHP code (see Tweaking PHP code on page 12).
- Improving the performance of your SQL queries since, for larger applications, PHP often is used in conjunction with a database (see Optimizing database queries on page 17).
- Making your PHP code run faster and references for further reading (see Other performance enhancements on page 21).

Background and motivation

For a recent consulting engagement, I reviewed a Web application nearing completion. The application was written in PHP, and had some serious performance problems. After a brief look at the code, I decided to search the Internet for any articles or information on tuning the performance of PHP code. To my surprise, I found very little information available. The PHP manual (see Resources on page 24) suggested simply throwing in more hardware. In my client's case, this was not always an option. Many systems were already in the field, and this application was a product upgrade. It was not feasible to upgrade every system in the field for this relatively minor product upgrade.

I had to devise my own tests to find ways of improving PHP. Thanks to IBM's *developerWorks*, I can share these findings with a much broader audience.

Tools

You can read through this tutorial from beginning to end and pick up a wide variety of tips and advice on how to improve your PHP code without having anything in particular installed.

For the most value, you may want to construct and run some of your own test cases. To do this, you will need a working installation of PHP on your machine. You should probably have the following:

- Any *OS supported by PHP* (http://www.php.net/manual/en/installation.php).
- *PHP (http://www.php.net/)*
- A Web server under which you run PHP is optional. Alternatively, you can run PHP from the command line.

Getting help

For technical questions about PHP, visit the *php.net* Web site. Read the *PHP manual* and pay particular attention to the user-contributed notes at the bottom of each manual page. They often contain valuable lessons learned that may help answer your question.

Refer to Resources on page 24 at the end of this tutorial for these and other useful links.

About the author

Steven Gould is a principal consultant with a large, international IT consulting and professional services company. Based in Dallas, he is a systems architect and senior developer, focusing primarily on Java and C++ development under Windows, Linux, and various UNIX platforms. Having worked on a variety of Internet and intranet applications, he also is very well versed in many other Internet-related technologies.

Tip: You also might be interested in the PHP Quick Reference Card available for free from *the author's Web site*.

Acknowledgements

I'd like to thank Shari L. Jones for her help reviewing this tutorial, and her unrelentless patience with me during the writing of it.

Additionally, I'd like to thank Akmal Chaudhri (editor) and IBM *developerWorks* for giving me the opportunity to write this much needed tutorial.

Section 2. Writing efficient code

Coding for efficiency

In this section, I review a variety of things that contribute to writing efficient code, whether in PHP or another language. Note that some of the following suggestions may, in the minds of some people, reduce the readability of the code. However, the payoff is in terms of improved performance.

You, as the developer, determine the points in an application where performance or readability and maintainability are the primary concerns. For example, if a piece of code is executed once per instance of an application, then readability is probably most important. However, for a piece of code that may be executed thousands, even millions of times per application instance, then performance is probably of greater importance than readability. Even this depends on a number of external factors -- such as the time available to execute that piece of the application, the amount of work to be done during each iteration, and the users' perceived performance of that part of the application.

If you do apply any of these techniques for improved performance, and you think it negatively impacts the readability of the code, add extra comments to the code and explain what is happening.

Optimizing loops

Looping, in various guises, is a fundamental concept in any programming language. It is very powerful, but can also cause performance problems in your programs.

Be careful which code you put inside a loop, especially if you expect the loop to be iterated through a large number (say, thousands) of times. In the next few panels, I look at several areas that require special attention when you use looping, whether the loops are `for`, `foreach`, `while-do`, `do-while` or any other form of loop. Broadly speaking, I consider each of the following:

- Eliminating redundant function calls within loops.
- *Inlining* function calls within loops.
- Creating and destroying objects within loops.

Even though the focus is on the preceding types of optimizations within loops, the gains from any of the performance improvement techniques discussed in this tutorial will be multiplied by any large loops that are impacted.

Tip: You probably have tens, even hundreds of loops in your application. To identify which ones to focus your efforts on, insert some simple debug that outputs an identifying line each iteration. That way you can tell, for a variety of different sets of input data, just how many times each loop is executed. You might be surprised at the results!

Optimizing loops: eliminate redundant function calls

Function calls can be very expensive in terms of CPU usage. Try to reduce the use of them inside loops. The two primary ways to do this are:

- Eliminate repeated calls to the same function with the same parameters
- *Inline* small, simple functions

I'll look at inlining separately in Optimizing loops: inline function calls on page 6 .

At first you may think, "Why would anyone repeatedly call the same function with the same parameters?" It's a reasonable question, but consider the following very common type of loop construct:

```
for ( $i=0; $i<count($myArray); ++$i )
   // do something
```

Although it appears innocent enough, that call to `count($myArray)` is made once every iteration. In any such loop, provided it doesn't add or remove elements from `$myArray`, the size of the array and hence the value returned by `count($myArray)` remains constant. You can write it more efficiently as follows:

```
$arraySize = count($myArray);
for ( $i=0; $i<$arraySize; ++$i )
   // do something
```

Notice how the call to `count($myArray)` is made just once now rather than once per iteration in the initial code example. This second approach does create another, temporary variable, but it gives better overall performance.

Optimizing loops: eliminate redundant function calls (continued)

In addition to the previous common trap (see Optimizing loops: eliminate redundant function calls on page 5 , also watch for code similar to the following:

```
$myConstString = "This is some constant text";
for ( $i=0; $i<$arraySize; ++$i )
   {
   print trim(strtolower($myConstString));
   // do something
   }
```

Provided the above loop does not modify `$myConstString`, then the call to `trim(strtolower($myConstString))` will always return the same value (in this instance, a string). You can move this function call to just before the loop, and store the return value in a temporary variable.

One other thing to notice with this last example is that ordering of function calls can impact performance. For example, trimming white space off of a string then converting the result to lowercase is marginally more efficient than converting the same string to lowercase, then trimming off any white space. This is because `trim` may shorten the string, therefore resulting in less work for the `strtolower` function. Either way, the result should be the same in both cases. However, trimming first potentially results in less computational effort.

Watch for similar optimizations in your code.

Optimizing loops: inline function calls

Often functions are defined that contain just one or two lines of code in the body. In this case, the overhead associated with calling and returning from the function may well be large compared to the actual work done by the function. Ideally, the compiler would detect this and replace the call to the function with the function body. Unfortunately, this rarely happens.

Languages such as C and C++ have an `inline` keyword to assist the compiler with this type of optimization. With PHP, however, it is necessary to perform this type of optimization manually. For example, consider the following code fragment:

```
function average( $a, $b )
   { return ($a + $b) / 2; }
...
for ( $i=0; $i<$arraySize; ++$i )
   $av[$i] = average( $array1[$i], $array2[$i] );
```

A more efficient implementation, albeit a little less desirable from a design and re-use perspective, moves the calculation out of the function `average` and into the caller. The result is:

```
for ( $i=0; $i<$arraySize; ++$i )
   $av[$i] = ( $array1[$i] + $array2[$i] ) / 2;
```

This second implementation eliminates the overhead associated with invoking the function `average`, and returning the result.

Only use this optimization for small functions and in places where the function could be called thousands of times. If you use this optimization technique for larger functions (where the overhead associated with invoking the function is dwarfed by the body of the function), the gains are minimal at the cost of far less readable and maintainable code.

Optimizing loops: create or destroy objects

Although PHP is not a fully object-oriented language, it does support the concept of objects. As with any language that supports object creation, a small overhead is associated with creating a new instance of an object, as well as with destroying an object. Therefore, create and destroy objects within a loop only when necessary.

Often, you can create an object outside of the loop. Then you can either treat it as read-only within the loop or, if different object settings are required in each iteration, modify its properties instead of destroying one object and creating a new one. For example:

```
for ( $i=0; $i<$arraySize; ++$i )
   {
   $pt = new Point( $i, $myArray[$i] );
   // do something with $pt
   }
```

In this case, assume a `Point` object has two properties of interest: `x` and `y`. In each iteration of the loop, a new Point object is created (with the `new` keyword) and destroyed. As an alternative, you can create one instance before the loop and then modify its properties in each iteration. This removes the overhead of destroying the old instance and creating a new one. The code would be as follows:

```
$pt = new Point();
for ( $i=0; $i<$arraySize; ++$i )
   {
   $pt->x = $i;
   $pt->y = $myArray[$i];

   // do something with $pt
   }
```

Note that because in PHP all of an object's member variables are public, this type of optimization should usually be possible. It may not always be desirable from a pure object-oriented design perspective, but that tradeoff is up to you as the developer.

Eliminating empty functions

Occasionally programmers new to object-oriented development and PHP may define subclasses that contain empty function definitions. This is necessary only if you intend to disable the behavior of a parent class -- generally as a result of a poor design in the first place.

Eliminate any empty functions, such as functions that simply return, and their function calls. The unnecessary overhead of calling a function that does nothing and then returning from it is eliminated.

Eliminating redundant code

When modifying existing code, be careful to remove any redundant code. For example, when I worked on code that several previous developers had modified, I found assignments to a variable that were always executed, but the resulting variable was never used. The problem is compounded if the variable is initialized by a call to a function, and the result is never used.

Avoiding comments that fail to add value

Some developers have a habit of adding a short comment consisting of just their initials, or sometimes their initials and the date. These comments rarely add any value to the code for anyone but the programmer who inserted them. Additionally, do not consider such comments *documentation*.

If you need a comment to explain the code, add information that will be helpful to other developers. If not, leave the comment out. It clutters up the code and potentially adds a small overhead in any scripting language.

Reordering switch...case statements

When using a switch...case statement, try to place the most frequently-occurring cases at the top of the list. This improves speed and efficiency for the most common cases. For example:

```
switch ( $user->writingHand )
   {
   case 'right-handed':
      print 'User is right handed'
      break;
   case 'left-handed':
      print 'User is left handed'
      break;
   default:
      print 'Not sure whether the user is left or right handed!'
   }
```

Handling special cases early

If your code includes any special cases where, for example, no action needs to occur in a function or method, place them as early in the code as possible and order them from most-likely-to-occur to least- likely-to-occur.

For example, in the function `ConvertToUpperCase` shown below, note that the test for an empty string is coded first. Then error conditions, such as the end index being less than 0 or the end index being less than the start index, are coded after that.

```
function ConvertToUpperCase( $str, $start, $end )
   {
   if ( $str == "" )
      return "";

   if ( $end < 0  ||  $end < $start )
      return $str;

   if ( $start < 0 )
      $start = 0;

   $len = strlen( $str );
   if ( $end > $len )
      $end = $len;

   return substr($str,0,$start)
          . strtoupper(substr($str,$start,$end+1-$start))
          . substr($str,$end+1);
   }
```

Using multi-dimension arrays with caution

Try to avoid excessive use of arrays, especially arrays of arrays, unless you really are handling data with two or more dimensions. For example, avoid using `$array[0][...]` to store one set of data such as the names of the months in a year, and `$array[1][...]` to store another set of loosely related information, such as sales data for each month. From a performance standpoint, it is more efficient to use two arrays such as `$array1[...]` (storing the names of the months) and `$array2[...]` (storing the sales data for each month).

Watching your network traffic

During one performance assessment, I noticed that a single user request to preview a report, being generated by PHP using PDFLib, actually generated two sequential HTTP GET requests. (The browser used was Microsoft Internet Explorer (IE) 5.5 SP1 -- the preferred browser at that time -- running on Windows 2000.) The resulting PHP script was actually running twice with the same inputs. I verified this by setting up a sniffer between the client machine and the server that reported exactly what was being sent back and forth between the two machines.

From the users' perspective, they would kick off the request to view a report. After some period of time -- actually after the script had run once -- the browser window would go blank. Then, after another equally long period of time -- after the script had completed for the second time -- the browser window would display the requested report. By eliminating one of these requests, I could cut the time that the user waits by half - a 100 percent improvement in performance.

I found that the problem of two GET requests being made to view a single report was specific to Internet Explorer. After researching the Microsoft Web site, I determined that multiple GET requests are *normal* behavior for Internet Explorer when it handles MIME-types such as PDF files. Based on information available from a Microsoft Support WebCast, "*MIME-Type Handling in Microsoft Internet Explorer* ," you can avoid two GET requests by ensuring that correct and complete headers describing the content are returned. In particular, you should include the following header:

```
Content-Type: application/pdf
Content-Disposition: inline; filename=report.pdf
```

By examining network traffic, I observed that the `Content-Disposition` line was omitted from the HTTP response header. After modifying the application to output this extra header, Internet Explorer correctly issued a single HTTP GET request for each report.

Carefully examine the network traffic, and differences between network traffic generated when using Internet Explorer, Opera, or Netscape. You might achieve a 100 percent improvement in performance for the recommended browser.

Returning content when it's available

In the same performance assessment mentioned in Watching your network traffic on page 9 , no data was sent back to the client browser until the entire PDF report had been generated. This was also observed by watching network traffic. When using PDFLib, you can generate PDF data directly in memory (instead of in a temporary file. Then, you can send each page of the report to the client as it is generated. The benefits of this method include:

- Network performance is better, especially for larger reports or over slower network connections. The first page can be sent before the entire report has been generated. As soon as the first page is ready, it is sent back to the client.
- Memory requirements on the server are reduced.
- The need for temporary files is avoided.

By making a few small changes to the application, you can implement this streaming of the reports as they are generated.

Ideally, the first page of the report is displayed as soon as it is received by the client browser. Limitations of the PDF format in use make this impossible. Adobe Acrobat Reader does support a Web-optimized PDF format that is more suitable in this instance. As of this assessment, PDFLib does not support output in this Web-optimized format.

Tip: Be careful not to send content back in blocks that are too small. You don't want to generate a large amount of extra network traffic. In the application mentioned above, sending back one PDF page at a time was a reasonable compromise between frequency of data sent back to the client and size of the data returned.

Section summary

In this section, you examined a variety of ways to improve the performance of your PHP applications. Although the suggestions used PHP code to illustrate how to improve performance of the code, the concepts presented here are largely language independent. You can apply many of the same concepts to improve the performance of applications developed in other languages.

In Tweaking PHP code on page 12 , let's look at performance improvements that are specific to the PHP language.

Section 3. Tweaking PHP code

The finer points of efficient PHP coding

In Writing efficient code on page 4 , you looked at several ways to write efficient code largely independent of the language you're using. Although illustrated with PHP examples, many of the same techniques could be applied to other languages such as C++ or Java.

In this section, I examine a variety of performance improvements that are more specific to PHP or exploit PHP-specific functions and behavior.

Using pre-increment versus post-increment

In PHP, as in C, C++ and Java, you can increment an integer variable by using the special unary operator: ++. For example:

```
$x = 0;
$x++;    // Post-incrementation. x is now 1
++$x;    // Pre-incrementation.  x is now 2
```

Similarly, you can decrement a variable by using the unary operator: `--`.

According to the *Zend Optimizer Technical FAQ*, pre-incrementation is a faster operation than post-incrementation. This is one of the simpler optimizations performed by the Zend Optimizer (see Resources on page 24 for details).

Therefore, it is recommended that you use pre-incrementation rather than post-incrementation whenever possible. The only place where this may make a difference is if the incrementation is done as part of an expression evaluation. If it is a stand-alone statement as shown in the preceding examples, you should have no problem changing from post-incrementation to pre-incrementation.

Similarly, you should use pre-decrement instead of post-decrement whenever possible.

Quoting strings

The PHP language supports two ways of quoting strings -- with double quotes or single quotes. Strings in are single quotes are not expanded but used as is. Often this is sufficient. See "Using Strings," listed in Resources on page 24 , for more information on the different uses of strings in PHP.

When using character strings don't need to be expanded, such as those without any variable references or escaped characters, use single quotes instead of double quotes. Because strings within single quotes are not parsed, they are executed more quickly than strings inside double quotes that must be parsed first.

Note that this performance enhancement may seem unnatural to C, C++ or Java developers who typically use double quotes for strings and single quotes to indicate a character. Since I work mostly in C++ and Java, I still find myself using double quotes for strings when writing PHP code. It's a hard habit to kick, but kicking it can buy you a little better performance from your PHP applications.

Understanding the difference between print and printf

PHP developers with a background in C and C++ may frequently be tempted to use the *`printf`* function to output strings. While this works fine, it does cost more in terms of CPU usage than the similar, yet more basic *`print`* function. Strictly speaking `print` is not a real function but a language construct.

Don't use `printf` when `print` is sufficient. To output simple strings, use `print`. Generally, you should only use `printf` (and `sprintf`) when you need more control over the output format, for example, when formatting integer or floating point numbers for display.

Understanding the difference between require and include

According to the *PHP manual*, `require` and `include` "are identical in every way except how they handle failure." However, further reading of the manual suggests another very subtle difference that impacts performance.

When you use the `require` keyword, the named file is read in, parsed, and compiled when the file using the `require` keyword is compiled. When a file containing the `include` keyword is compiled, the named file is not read in, parsed, and compiled initially. Only when that line of code is executed is the file read, parsed and compiled.

Only use the `require` keyword if you know you will *always* need that named file in the current script. If you *might* use its functions, use `include` instead. PHP opens up all files that are required, but only opens included files as needed.

Additionally, you should also consider using `require_once` and `include_once` in place of `require` and `include` respectively. In practice, it is more likely that you actually want the functionality provided by the `require_once` and `include_once` functions, even though it is much more common to use the `require` and `include` keywords respectively.

Refer to the following PHP manual pages for more information: *include*, *include_once*, *require*, *require_once*.

Keeping source include files small

Since, PHP files are parsed and compiled at runtime, you should try to keep your source file sizes as small as possible. Source file size is particularly important with files that are included by other files. Only `include` or `require` files you absolutely must have. Additionally, if you only need one function in a file, consider breaking that function into its own file.

One thing of interest here is that after running tests on a variety of source files with varying amounts of comments, I found that the amount of comments in a source file had no significant impact on performance. The only thing that seemed to affect performance was the amount of PHP code in the files. This seems reasonable when you consider that comments basically are ignored by the compiler, while any code must be parsed and compiled -- something that is much more time consuming.

Making effective use of built-in functions

PHP has a very rich set of built-in functions. In general, using an appropriate built-in function is more efficient than trying to implement the same functionality in PHP manually.

An example of this is the `date` function. I have seen several instances where a developer has tried to output a date by first constructing an array with the `getdate` function. Then, to generate the required output, the developer accesses the various elements of the array.

In this instance, the developer could have used the `date` function to format date and time strings with just one call to a built-in function. When you want to output date strings, use the power of the `date` function instead of trying to code the logic manually.

Similarly, when implementing code to perform a fairly common task, take a few minutes to review the PHP manual. You might find a built-in function that you can use instead.

Reducing looping with built-in functions

The built-in functions of PHP can dramatically improve the efficiency of your code in the area of looping. When you encounter a block of PHP code that iteratively calls a built-in function, take time to see if a more efficient way of achieving the same task is available through a different, yet related, built-in function.

For example, it's faster to prepare a large string with UIDs and call `imap_fetch_overview` once than it is to call `imap_fetch_overview` repeatedly in a loop. See the user-contributed comments for the *imap_fetch_overview* function in the PHP manual for more details.

Similarly, when reading large files it is far more efficient to read, say, a line at a time (using `fgets`) and process that, than it is to read and process a character at a time (using `fgetc`). There are many other similar cases throughout the PHP language, depending on exactly what your application does.

Use HTTP 1.0 with fsockopen

The PHP function `fsockopen` initiates a stream connection to another machine across a network. When you use this to communicate with another machine, use the simpler HTTP 1.0 protocol unless you specifically need some of the features provided by the HTTP 1.1 protocol.

With HTTP 1.1, the default behavior is to keep connections alive until they are explicitly closed or the connection times out. This behavior can create some serious performance consequences if clients that don't expect, or utilize the behavior. (Note that for clients designed with this behavior in mind, keeping connections alive actually improves performance.) Unless your application requires that connections be kept alive, then seriously consider using the HTTP 1.0 protocol with `fsockopen`, as in the sample code below:

```
$fp = fsockopen( $server, $port );
fputs($fp, "GET $page HTTP/1.0\r\nHost: $server\r\n\r\n");
```

When using the HTTP 1.1 protocol to make a request to another machine, you may also encounter *unexpected* hex values interspersed amongst the returned data. These unexpected hex values are the result of a *chunked* transfer-encoding which, according to the HTTP 1.1 specifications must be supported by any HTTP 1.1 application. Therefore, your application should either handle this type of encoding, or not claim to be an HTTP 1.1 client.

So, in summary, claim to be an HTTP 1.0 client unless you specifically need features of the HTTP 1.1 protocol. If you do need HTTP 1.1 features, then be sure to support the newer protocol fully.

Avoid the "*Top 21 PHP Programming Mistakes*"

Finally, an interesting and useful three part series, "Top 21 PHP Programming Mistakes," is available on the Zend.com Web site. While not all of the listed mistakes directly impact performance, it is a good idea to familiarize yourself with them and try to avoid them.

- *Part 1: Seven Textbook Mistakes*
- *Part 2: Seven Serious Mistakes*
- *Part 3: Seven Deadly Mistakes*

Section summary

You have just looked at improving the efficiency of your PHP applications by using features of the PHP language to your advantage. Of these, you can probably get the most significant improvement through careful selection of the most appropriate built-in function for the task at hand. In fact, this idea alone constitutes nearly half of this section. You can use the other techniques discussed to further tune your applications for best performance.

In Optimizing database queries on page 17 , let's look at improving the performance of your SQL code when used within your PHP applications.

Section 4. Optimizing database queries

The importance of efficient SQL

Most Web applications of any size involve the use of a database. Typically, a Web application allows the addition or creation of new records (for example, when a new user registers on the site), and the reading and searching of many records in a database. The most common bottleneck when developing a Web application is in the reading of a large number of records from a database, or executing a particularly complex `SELECT` statement against the database.

Writing to or updating a database usually is performed on a small number of records at a time. This is often much less of an issue than cases that involve reading thousands of records at a time. Consequently, in this section, I look at different ways to speed up your database queries (also known as reads). Many of the same techniques can be applied to database updates (also known as writes).

Finally, since I'm looking at ways to write more efficient SQL statements, the techniques presented here are independent of the underlying database engine. Obviously, you will obtain different performance improvements with different database engines, as well as different database schema.

Streamlining SELECT statements

Review any embedded `SELECT` statements to identify any unused fields. That is, identify fields that are retrieved from the database but whose values are never used in the code following the query. This situation often results from code that has passed through the hands of several developers, or that has undergone several iterations of changes.

By eliminating unused fields from `SELECT` statements, you reduce the complexity of the query and reduce the amount of data sent over the network (or at least between the database server and the PHP script). The net affect of making such changes is a reduced database read time.

Eliminating any unnecessary ORDER BY clauses

You can also reduce database read time by eliminating unnecessary `ORDER BY` clauses in your SQL queries. For example, consider the following `SELECT` statement designed to retrieve the product name and price when given a product number.

```
SELECT product_name, price
FROM catalog
```

```
WHERE product_num = "123456789"
ORDER BY product_name
```

Assuming the product number is unique for all products in the catalog, this query only returns a maximum of one record. Even though sorting 0 or 1 records should be very quick, the `ORDER BY` clause is actually redundant. When this query is executed hundreds or thousands of times for different products, this redundant overhead can add up. Eliminate this type of redundancy.

Watching for complex SELECTs

The following shows the outline of a complex `SELECT` statement to be executed:

```
SELECT ... FROM ...
WHERE cond AND field IN
  (
  SELECT field1
  FROM table2
  WHERE cond
  GROUP BY ...
  HAVING ...
  )
GROUP BY ...
ORDER BY ...
```

The exact fields, tables, and conditions are not important as they are usually determined by the business. What is important to notice, however, is the use of the IN clause with another internal `SELECT` statement. This can prove to be very expensive in terms of query execution time. Depending on your specific data set, a much faster approach may be to create a temporary table holding the results from the internal `SELECT`, then use a greatly simplified `SELECT` statement within the IN clause. For example:

```
SELECT ... FROM ...
WHERE cond AND field IN
  ( SELECT field1 FROM temp_table )
GROUP BY ...
ORDER BY ...
```

Now both `SELECT` statements execute many times faster than the previous single `SELECT` statement.

In some tests that I ran on a client's data set, I even found that the percentage of reduction in query execution time appeared to increase as the size of the data set was increased. The largest data sets, with the longest execution times, realized the largest percentage of time reduction.

Eliminating or reducing duplicate requests for the

same data

When you process many records from a database, especially those cross-referenced with another database, try to modify the code to eliminate -- or at least reduce -- duplicate requests for the same data.

For example, you generate a detailed sales report from a database that contains only product numbers and actual sales. In the report you also need to display the corresponding product names that are maintained in a different database. You may be able to implement a basic caching mechanism so the product names are only retrieved once for each product number.

This approach will consume more memory. But, depending on your specific data set, it also can make a significant impact on the total amount of time spent reading data.

Considering indexes

With most database engines, you can execute queries manually at the equivalent of a command-line. The database engine will show you which indexes, if any, it used when executing the query. Try this with some of your longer running queries. You may be surprised at the results.

In some cases, even though various indexes may exist on a table, the database engine may not be able to use them for your particular type of query. Consider the pros and cons of adding another index to this table.

As a general rule of thumb, try to create simple indexes (indexes on just one field in a table). Simple indexes are often more useful to the query engine than more complex, compound indexes. Additionally, the simple indexes typically require less maintenance overhead for the database engine used.

Handling results sets

Assuming you've done all you can to optimize your database queries, the next thing to look at is how you access the results within your PHP code.

For now, I assume that you are using the Unified ODBC functions in PHP since these are the most generic. The same concepts apply even if you are using one of the other database-specific sets of functions.

Rather than using something like `odbc_result` to retrieve individual fields, use `odbc_fetch_into` to get the entire row into an array then access the array directly. This results in far fewer functions calls especially when iterating over a large record set.

Additionally, when accessing array elements, use numbered indexes rather than named indexes. Although these generally are less readable, they result in faster runtime execution. To address the readability issue, add short comments to indicate which fields are accessed in each case.

Finally, avoid repeated calls (for example, when processing individual records or rows) to the functions such as `odbc_field_name`, `odbc_field_type`, `odbc_field_len` and other `odbc_field_*` functions. Such functions return the same value for all records in a result set. The number of fields are always the same within a given query, as are the field names, and the type and length of each field. Therefore, if any of these values are required, retrieve them once after making the initial query/ Then, save the result in a local variable that can be referenced as needed for each record.

Section summary

Now, you have looked at a variety of ways to improve the performance of database queries in your PHP applications. Since the usual performance bottleneck is the reading of large quantities of data, the focus of this section was on database reads or `SELECT`s. However, you can apply many of the same concepts to alleviate any performance bottlenecks associated with database updates.

In addition to reviewing the SQL statements used to extract data, you should also be conscious of performance issues that may have been introduced by how you retrieved the result sets.

In Other performance enhancements on page 21 , I go on to look at more drastic ways to improve the performance of your Web application.

Section 5. Other performance enhancements

The business need for speed

The suggestions described in this section are more general than coding improvements and generally have a larger business impact. Each should be considered with respect to your specific business conditions. There may be very good business reasons why you are not able to take advantage of some or all of these. For example, upgrading your servers may be a simple task if you only have one or two servers and they are easily accessible. However, if your application is installed on hundreds or thousands of your customers' servers, performing a hardware upgrade may be impractical or infeasible.

Still, it is worthwhile revisiting these options periodically, possibly when starting a major new release of the application.

Running PHP as a module

If you currently run PHP as a CGI executable (the default), and you use a Web server for which a PHP module available is available (such as Apache), consider reconfiguring the server to run PHP as a module. This avoids the overhead associated with starting and terminating a CGI process for each client request. This modification has the greatest impact when serving a large number of requests that involve short scripts, as opposed to relatively few requests that involve much longer scripts.

To find out if there is a PHP module available for your particular Web server, or for more details on configuring your Web server to use a PHP module, refer to the *installation section* of the PHP manual.

Using the APCs of PHP

According to the *Alternative PHP Cache (APC) Web site*:

"APC was conceived of to provide a way of boosting the performance of PHP on heavily loaded sites by providing a way for scripts to be cached in a compiled state, so that the overhead of parsing and compiling can be almost completely eliminated."

Consider the use of the Alternative PHP Cache (APC) to help improve the performance of your Web application. It is open-source and free, and could loosely be described as an open-source competitor to the Zend Optimizer (discussed in Considering the use of the Zend Optimizer on page 22).

Considering the use of the Zend Optimizer

Zend, the primary developers of PHP, offers another pair of products that together can help improve performance:

- *Zend Encoder*, a product that encodes your PHP scripts to protect your PHP scripts against reverse engineering.
- *Zend Optimizer*, a product that works with the files generated by the Zend Encoder and applies multi-pass optimizations to help speed up the execution of your PHP code.

If your application is not currently running under a platform supported by the Zend Encoder and Zend Optimizer, consider moving to a platform supported by these products. While the Zend Optimizer is free, the Zend Encoder is not. You will also need to weigh the cost of this product against the benefits achieved when using it in conjunction with the Zend Optimizer.

Rewriting core functionality in C/C++

A more extreme approach to improving performance would be to rewrite some of the larger and most frequently-used functions and methods in a compiled language such as C or C++. If you can generalize this functionality and feel it would be beneficial to others, submit it for inclusion in the core PHP language or as an extension. This actually is how some of the built-in PHP functions came about.

Reviewing dynamic generation of PDF files

If your application dynamically produces PDF files using ClibPDF, PDFLib or similar, consider the following:

- Is a proportional or fixed font appropriate for the information?
  When you use a proportional font, correct alignment of numbers requires calls to functions such as `pdf_stringwidth` to get the width of a number to compute the coordinates of the left-hand side of the number. Such calls often are made many times throughout the generation of a PDF file, and can add up. An alternative approach, though possibly less acceptable to users, would be to use a fixed font such as Courier. You can more easily keep track of the text positions within your own code.

- Is it absolutely necessary to output reports in PDF or is an HTML format sufficient?
  HTML can be generated much more quickly. Perhaps users would be happy with an HTML format report for on-line viewing and then, only if a printable or print-friendly report is required, generate a PDF file.

- Just how timely and dynamic do the these reports need to be?
  Reconsider the business case for having these reports. You might be able to batch generate some commonly-requested reports, perhaps nightly, and make them directly available through a hyperlink.

Other performance enhancements

More major suggestions for improving performance are given on the Zend Web site in the article, "Optimizing PHP Scripts" Resources on page 24). While this is a good article, its bias is that "programming time is expensive". This may be true, but as discussed earlier in Background and motivation on page 2 , there are cases when programming time is still the most viable option or the only option available. It also may be the only option open to you, the developer.

To summarize some of the main points from the Zend article, other options to consider when trying to improve performance include:

- Upgrade the hardware on which your PHP server is running.
- Upgrade the Operating System (OS). Zend suggests using a UNIX-based system such as Linux or BSD UNIX.
- Reconsider the choice of database. For example, if your application does not need stored procedures and sub-queries, then consider MySQL for improved performance.
- Consider the use of the Zend Optimizer (free) and Zend Cache. While these products support most of the major platforms, they currently are not available for all platforms on which PHP is supported.

Section summary

In this section, you looked at improving the performance of your PHP applications by re-designing parts of your application as well as the hardware on which it runs. In many commercial applications, these often require approval from the business owners due to additional expense for hardware and software, as well as additional time to complete the application. As a result, many of these options are not readily viable in a business environment.

Section 6. Summary

Wrapup

In this tutorial, you reviewed the following ways to improve the performance of your PHP applications:

- General guidelines for writing efficient code (largely language independent)
- PHP language-specific efficient programming tips
- Writing efficient SQL queries
- Other means of improving performance

If you followed through the entire tutorial, you should have gained some insights about improving the performance of your PHP applications. If you'd like to more reading on any of these techniques, refer to Resources on page 24 .

Resources

Unfortunately, relatively little information is available about writing efficient PHP applications -- hence the motivation for writing this tutorial. The following are some resources that I used in writing this tutorial.

- Visit the *PHP* Web site (http://www.php.net).
- If you have problems with a particular function, read the user-contributed notes and comments throughout the *PHP manual* (http://www.php.net/manual/en/). Chances are good that someone else has experienced the same problem and added notes about it to the manual.
- The article, *Optimizing PHP Scripts* suggests "programming time is expensive." Read how to improve the performance of your PHP applications from other angles. It also contains a couple of useful sections on *when* to optimize, and how to measure performance of pieces of your application.
- Check out *Zend*, the commercial side of PHP. It provides additional products that can help improve performance: *Zend Cache* and *Zend Optimizer*.
- See "*Using Strings*" on the Zend Web site for more information on the different uses of strings in PHP.
- Get further information on Internet Explorer's handling of MIME-types from Microsoft Support WebCast: *MIME-Type Handling in Microsoft Internet Explorer* , January 11, 2001. In particular, refer to the Questions & Answers section at the end of the Web cast where several questions address the possible need for two GET requests.
- Learn how to make your wireless content more dynamic with the use of PHP in the IBM *developerWorks* tutorial, *Flex your PHP* (*developerWorks*, March 2002).
- Visit *Steven Gould's Web site* and get the useful PHP Quick Reference Card.

Your feedback

We welcome your feedback on this tutorial, and look forward to hearing from you. We'd also like to hear about other tutorial topics you'd like to see covered.

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.