

Knowledgebase (KB)






NeoSOFT®
TECHNOLOGIES

All Categories

-   HR Policies (40)
-   Employee Performance Management System (0)
-   Training (16)
-   Utilities (2)
-   Knowledge Sharing (15)
 -   Mobile (0)
 -   Others / Misc (6)
 -   Web (3)
 -  Database Guidelines
 -  **Hardware - Software Performance Guidelines**
 -  Hosting Guidelines
 -  HTML-CSS-JS Guidelines
 -  Neosoft Secure Coding Practices
 -  Security Guidelines

There are no categories to display in knowledge base.

Recently Viewed

-  Database Guidelines
-  Business Conduct And Ethics
-  Working Hours
-  Workplace Policies
-  Guidelines for Working in Night Shifts

 [Home](#) » [Group Categories](#) » [Knowledge Sharing](#)

Hardware - Software Performance Guidelines

Article Number: 123 | Rating: 4.5/5 from 2 votes | Last Updated: Wed, Mar 30, 2016 at 1:15 PM

Architecture: Choosing the right technologies from performance standpoint

- Database Selection
 - Relational vs NoSQL
- Jobs / Queues / Asynchronous Processing
- Backend Technology Selection

OS / Server/ Hardware Optimization

- Disable last accessed time function of filesystem on Database Server. Do this with care, as this does have security implications.
- Use high performance file system such as EXT4 or BRFS
- Recommend the end client to use SSD or high performance IO. (PS: Deleted data cannot be recovered from SSD, please ensure timely backup when moving to SSD)

Monitoring

- Use monitoring tools such as NewRelic to monitor and identify performance issues
- Setup automated alerts relevant team members to improve incident response

Database Performance Tuning

- Use persistent connections when the Database server is on different server and connection latency is high and when you are not expecting lot of simultaneous users (such as intranet applications). Read more: <http://php.net/manual/en/features.persistent-connections.php>
- User asynchronous logging
- Use indexes on all fields referred in joins and other read heavy fields. Beware that joins reduce write performance, so use carefully on write-driven applications.
- Use EXPLAIN to understand performance implications of SQL Queries
- If you application has heavy unavoidable joins. Use in-memory swap space to improve join performance.
- Upgrade to latest version of DB server
- Query Optimization
 - Use profiler

Backend Code Guidelines

- Minimize the number of notices and warnings thrown by the PHP code to reduce logging overhead
- Don't run synchronous operations in for loop. Example: Don't run MySQL Queries in loop or send CURL requests in loop
- Use indexes on all fields referred in joins and other read heavy fields. Beware that joins reduce write performance, so use carefully on write-driven applications.
- Official PHP Guidelines
 - <http://php.net/manual/en/features.gc.performance-considerations.php>
- Queues, Messaging and Event Systems
 - Use queues to asynchronously offload time consuming operations for example:
 - Image resizing and optimization
 - File transfer
 - Frameworks such as Laravel have inbuilt features/classes to provide this functionality.
- Enable PHP Profiler in xDebug on development and staging environments to understand the slow parts of the apps
- User CURL over file_get_contents()

Web Server / CGI Handler Guidelines

- Prefer Nginx over Apache
- Use PHP FPM and Latest version of PHP
- Enable compression(such as GZip) for appropriate MIME types
- Headers
 - Keep Alive
 - Caching
 - ETags
 - Expires
 - FileTag: None header
- Use appropriate session cookie expiry time
- If using PHP 5.4 or higher enable opcache, increase cache file limit. Otherwise use APC/xCache. Use commercial PHP accelerator tools whenever appropriate.
- Minimize redirects (30X redirects)
- Optimize php.ini parameters as per your App's requirements
 - Ensure datetime.timezone is defined

Caching / CDNs

- Use separate cookie-free domain
- Use subdomains to serve images and other asynchronous content
- Server/Fronend Caching (e.g. Varnish)
- Query/Data Caching (Memcache, Redis)
- HTTPS
 - Use HTTP/2 or SPDY if application is SSL enabled. Free SSL certificates are availalbe from: <https://letsencrypt.org/>



- These protocol keep the connection between server and client open and allow all request to be served within single handshake
- CDN (Cloudflare, BitGravity)
 - Use services like cloudflare as frontend caching and HTTPS endpoint
 - Load following things from CDN
 - Scripts & Stylesheets such as libraries jQuery
 - Images
 - Large binary/zip/executable file downloads
 - Publicly available documents, PDF, DOC etc.

Front-End Guidelines

- Use and follow guidelines from GTMatrix, Google PageSpeed and YSlow
- Combine and minify JS & CSS Files (use [grunt](#) or <http://mun.ee/>)
- Use Sprites instead of separate images
- Minify HTML using a tool like tidyHTML
- CSS links at the top of the page, JS, especially external JS should be placed at the bottom
- Use optimum compression and optimization for PNGs and JPEG files (Tool: <https://github.com/gruntjs/grunt-contrib-imagemin>)
- Specify image dimensions
- Defer parsing of JS when appropriate
- If appropriate, dynamically load resources that don't have code dependency such as advertisements.
- Avoid charset in META tag
- Avoid duplication: Of content, same stylesheets or scripts served from multiple URLs
- Specify HTML at the top of the source codes
- Use Ajax calls for heavy data after the core user experience is loaded on the web page. For example: Dashboards, Games or Large Reports
- Avoid CSS @import, use links
- Avoid using CSS filters
- Preload content:
- Prefer GET request for Ajax in

If a method can be static, declare it static. Speed improvement is by a factor of 4. `echo` is faster than `print`, use `echo`'s multiple parameters instead of string concatenation. Set the max value for your for-loops before and not as a function in the loop: `$len = count($big_array); for ($i=0; $i<$len; $i++) {...}` Unset your variables to free memory, especially large arrays.

Avoid *magic methods* like `__get()`, `__set()` and `__autoload()`

To find out the time when the script started executing, `$_SERVER['REQUEST_TIME']` is preferred to `time()`

See if you can use `strncasecmp`, `strbrk` and `stripos` instead of `regex`

See if you can use `is faster than`, `str_replace` by a factor of 4

It's better to use select statements (`switch()` { case ": } }) than multi if and else if statements. Error suppression with `@` is very slow.

Turn on apache's `mod_deflate`

Turn on apache's `mod_gzip` which is available as an Apache module, compresses your data on the fly and can reduce the data to transfer up to 80% Close your database connections when you're done with them (although PHP does it for you upon full page load or end of script execution)

Use single quotes in indexes: `$row['id']` is 7 times faster than `$row[id]`

Use `sprintf` instead of variables contained in double quotes, it's about 10x faster.

Incrementing a local variable in a method is the fastest. Nearly the same as calling a local variable in a function.

Incrementing a global variable is 2 times slower than a local var.

Incrementing a local variable in a method is the fastest. Nearly the same as calling a local variable in a function.

Incrementing an object property (eg. `$this->prop++`) is 3 times slower than a local variable.

Incrementing an undefined local variable is 9-10 times slower than a pre-initialized one.

Just declaring a global variable without using it in a function also slows things down (by the same amount as incrementing a local var). PHP does a pricy check to see if the global exists. Methods in derived classes run faster than ones defined in the base class.

A function call with one parameter and an empty function body takes about the same time as doing 7-8 `$localvar++` operations. A similar method call is of course about 15 `$localvar++` operations.

Surrounding your string by ' instead of " will make things interpret a little faster since php looks for variables inside "..." but not inside '...'.

When working with strings and you need to check that the string is of a certain length, you'd understandably would want to use the `strlen()` function. This function is pretty quick since it's operation does not perform any calculation but merely returns the already known length of a string available in the `zval` structure (internal C struct used to store variables in PHP). However because `strlen()` is a function it is still somewhat slow because the function call requires several operations such as lowercase & hashtable lookup followed by the execution of said function. In some instance you can improve the speed of your code by using an `isset()` trick.

```
if (!isset($foo{5})) { echo 'Foo is too short'; } ?>
```

instead of

```
if (strlen($foo) < 5) { echo 'Foo is too short'; } ?>
```

Calling `isset()` happens to be faster than `strlen()` because unlike `strlen()`, `isset()` is a language construct and not a function meaning that it's execution does not require function lookups and lowercase. You'll have no overhead on top of the actual code that determines the string's length.

When incrementing or decrementing the value of the variable `$i++` happens to be a much slower than `++$i`. It is because instead of 4 opcodes used for `$i++` you only need 3. Post incrementation actually causes the creation of a temporary var that is then incremented while pre-incrementation increases the original value directly. This is one of the optimization that opcode optimizer such as Zend's PHP optimizer utilizes.

Use `array_pad()` A common case when assigning multiple variables from an array is to use the `list()` construct with the `explode()` function to split the string and set the parts of the list to each variable like `so:list($foo1, $foo2, $foo3) = explode('/', 'var1/var2/var3');` However, the problem is then created where you may not be bringing in as many variables for each part of the list. If you are missing a variable or two when you explode, you will get a nice notice from PHP. The usual response to this is to use the error suppression operator to dismiss the error (`@list()`). In this case, `$foo3` will be set to `NULL` and your script will suffer a speed penalty for use of the `@` operator. To alleviate this problem it is better to use `array_pad` to only fill the array with the number of variables that you have instead of causing and suppressing an error at the cost of your performance.

```
list($foo1, $foo2, $foo3) = array_pad(explode('/', 'var1/var2'), 4, NULL);
```

Not everything has to be OOP, often it is too much overhead, each method and object call consumes a lot of memory.

[Excellent article](#) about optimizing your PHP code by John Lim

Do use `foreach` for looping collections/arrays. PHP4 items are byval, < PHP5 items are byref

Do consider using the [Singleton Pattern](#) as you will have only one instance of an object.

Do use POST over GET for all values that will wind up in the database for TCP/IP packet performance reasons.

Excellent article about optimizing your PHP code by John Lim

Do use `ctype_alnum` , `ctype_alpha` and `ctype_digit` over regular expression to test form value types for performance reasons.

Do use full file paths in production environment over `basename/ file_exists/ open_basedir` to avoid performance hits for the filesystem having to hunt through the file path. Once determined, `serialize` and/or cache path values in a `$_SETTINGS` array. `$_SETTINGS['cwd'] = cwd(./);`

Do use `require/ include` over `require_once / include_once` to ensure proper opcode caching.

Do use `tmpfile` or `tempnam` or creating temp files/ filenames

Do use a proxy to access web services (XML or JSOM) on foreign domains using XMLHTTP to avoid cross-domain errors. eg. `foo.com<-->XMLHTTP<-->bar.com`

Do use `error_reporting (E_ALL)`; during debug.

Do set Apache `allowoverride` o "none" to improve Apache performance in accessing files/directories.

Do use a fast fileserver for serving static content (`thttpd`) `static.mydomain.com`, `dynamic.mydomain.com`

Do serialize application settings like paths into an associative array and cache or serialize that array after first execution.

Do use `PHP output control` buffering for page caching of heavily accessed pages

Do use `PDO` (or a major PHP framework such as `CakePHP`, `Symfony`, `CodeIgniter`, `Yii` or `phpiphany`) for prepared statements over native DB prepare. `mysql_attr_direct_query=>1`
Do NOT use SQL wildcard selects. eg. `SELECT` (although selecting all columns manually yields same retrieval time as issuing `SELECT` ; therefore try to select only the columns you are actually going to use)

Do use database logic (queries, joins, views, procedures) over loopy PHP.

Use caching whenever possible. [This article](#) defines basic caching techniques. To sum it up, use arrays for caching big chunk of data in the memory directly. Use `APC` or `eAccelerator` on a single server. Use memcache on multiple servers when you scale you application.

APC being the fastest caching method does not scale well. `Redis` and `Memcache` can be used for cross-process/multi-server communications. Data in `Redis` storage will remain even after a server reboot. Don't want to install `Redis` (it's just a 1 line in console)? Use `Memcache`. If you can't install any third-party packages, you can use `Shared Memory` – but your `PHP` should be compiled with support of `shmop-functions`. If we take `APC` as our unit of measure in the following benchmark, we can draw the following conclusion:

- `APC` – best performance – speed 1
- `Redis` – speed 1.6
- `Shared memory` – speed 130
- `Memcache` – speed 192 (slowest)

Head over to the [Redis VS Memcache](#) comparison to see the stats in action.

Here is [another interesting resource](#) that provides comparison of most of the items listed above

Lastly, follow [Yahoo performance rules](#) for best practices on optimizing loading times of your website

Posted by: PHP - Administrator - Wed, Mar 30, 2016 at 1:15 PM This article has been viewed 317 times.

Filed Under: Knowledge Sharing

Article Rating (2 Votes)



You've Already Voted.

[Bookmark Article \(CTRL-D\)](#)

Attachments

There are no attachments for this article.

Related Articles

[Database Guidelines](#)
Viewed 448 times since Wed, Mar 30, 2016

[Security Guidelines](#)
Viewed 620 times since Wed, Mar 30, 2016

[HTML-CSS-JS Guidelines](#)
Viewed 445 times since Wed, Mar 30, 2016

[Hosting Guidelines](#)
Viewed 248 times since Wed, Mar 30, 2016

[Neosoft Secure Coding Practices](#)
Viewed 908 times since Wed, Mar 30, 2016