**Chapter 1. Drupal Coding for Optimal Performance**

One of the great things about Drupal is the ease with which you can extend or override core functionality in order to customize it for your specific needs. However, if you are not careful with how you code, you may introduce a huge performance bottleneck into your contributed or custom module. This chapter will give an overview of Drupal APIs relevant to performance and scalability, common coding best practices, and pitfalls to be aware of when trying to approach common tasks.

**Context Matters**

Before discussing the APIs and patterns that Drupal provides, it’s worth discussing which types of issues are often introduced when writing Drupal code.

Performance and scalability issues in code can affect CPU, memory, filesystem, database, and network usage, either individually or in combination. All code uses at least some CPU and memory, and all sites will access the database and filesystem and potentially make network requests. Whether any of these turns out to be a performance bottleneck is always down to context.

There are no hard and fast rules about what makes code "fast" or "slow"—exactly the same code could be acceptable in one situation but not in another, and performance often needs to be balanced against other programming issues such as testability, readability, and maintainability.

When writing or reviewing code, it’s important to think of the context the code will be executed in—both the immediate use case and whether it might also be applied to other contexts. The following are some general questions to ask, before you start trying to optimize at all:

* Does the code get executed on every request?
* Could it run more than once during a request? If so, a few times, or hundreds or thousands?
* If the code runs less frequently, will it affect end user performance? And how critical is end user performance in that case?
* Does the code have side effects that could affect the performance of other requests, such as writing to the database or flushing caches?
* Is the code an isolated unit, or will it be affected by other code or the configuration and state of the Drupal installation it runs on? For example, the amount of content, users, themes, or modules installed can dramatically change the characteristics of how code performs.

Only after considering these questions should you attempt to apply one or more of the approaches outlined here.

**False Optimizations**

It’s entirely possible to make the performance of code worse by "optimizing" it. This happens when additional code is added to avoid expensive processing, but the expensive processing happens anyway. The result is that both the original expensive code and the new code run, adding additional overhead to an already bad situation.

An example of this is the fairly common micro-optimization of replacing array\_key\_exists() with isset(). (Please note that this is used only as an example, and we’re not explicitly recommending doing so!):

isset()

This is a language construct that tells you whether a variable is set or not, and returns false if that variable is explicitly set to NULL.

array\_key\_exists()

This is a function that tells you if an array key exists regardless of the value.

Function calls in PHP have more overhead than language constructs, so isset() takes less time than a function call, and while the semantics are different, they can be used interchangeably if you don’t need to explicitly check for array keys set to NULL. Hence, a common micro-optimization is to use isset() unless it’s absolutely necessary to check for NULL.

Let’s assume you had some code that definitely needed to use array\_key\_exists() because of the NULLcheck, but you wanted to try to run the faster isset() first, to skip the function call when it’s not needed. You might write code like this:

<?php

$array = array('foo' => NULL);

isset($array['foo']); *// returns FALSE.*

array\_key\_exists('foo', $array); *// returns TRUE.*

isset($array['foo']) || array\_key\_exists('foo', $array); *// returns TRUE.*

?>

The last example is semantically identical to just an array\_key\_exists() call, but in the case that $array[‘foo’] is set to a non-NULL value, **only the isset() check needs to be made, avoiding the more expensive function call.**

However, if $array[‘foo’] doesn’t exist or is set to NULL, then the code actually has to do more work—checking isset() then the array\_key\_exists(), as well as the || operator—all of which is going to be slower than just running array\_key\_exists() in the first place!

The only way to know the effect of this is to create a realistic scenario or test on a real install, and see which code execution path is actually the most common. This comes back to context—it’s not so much the content of the code itself that determines its performance, but how exactly it is executed.

Whether this kind of optimization is a problem depends on the relative performance increase you hope to gain.

For example, when checking access rights, you may need to check an administrative permission via user\_access() as well as access permissions based on an entity ID, which requires loading the entity via entity\_load() first. Both checks are necessary regardless, but the order is important.

While very few users might have the administrative permission, a call to user\_access() takes a fraction of the resources that loading and access-checking an entity does and won’t cause a measurable delay. It’s worth doing the cheaper check first even if the second, more expensive check will run too.

This is the same with almost any pattern that attempts to circumvent code execution rather than completely rewriting it. For example, adding persistent caching to a function that is a cache miss in 99.9% of cases will mean extra time spent checking and writing to the cache, as well as extra space being taken up in cache storage, on top of the original code being executed. However, if the code being executed is very expensive, then the overhead of cache misses may well be outweighed regardless.

With this in mind, we’ll first cover a common task for Drupal custom and contributed modules, and look at ways to ensure that this task is executed as fast as possible. Then we’ll move on to the APIs that Drupal provides specifically to aid with performance and scaling.

**Listing Entities**

Whether it’s on the front page of a blog or in a gallery of images or a comment thread, much of the work done on a Drupal site involves getting a list of entities and then rendering them.

There are two APIs introduced in Drupal 7, and only slightly changed in Drupal 8, that help with this:entityQuery() and entity\_load\_multiple().

**entityQuery()**

Rather than a direct database query to entity and field tables, EntityQuery() relies on a storage controller to handle building and executing the query for the appropriate entity storage backend. This has the advantage that any query run through entityQuery() is storage agnostic, so if you’re writing a contributed module or working on a site where it might be necessary to move to alternative entity storage in the future, all your queries will transparently use the new storage backend without any refactoring. EntityQuery() can be used whether you’re writing queries by hand in custom code or via the entityQuery() Views backend.

**Multiple Entity Loading**

Once you have some entities to list, you’ll need to load and then render them.

A common pattern would be to loop over each node and load them individually:

<?php

/\*\*

\* Provide an array of rendered entities given the IDs.

\*

\* @param array $ids

\* The entity IDs to load

\*

\* @return $rendered\_entities

\* The array of rendered entities.

function **render\_entities**($ids) {

$rendered\_entities = array();

foreach ($ids as $id) {

$rendered\_entities[$id] = entity\_view(entity\_load($id));

}

return $rendered\_entities;

}

?>

Drupal 7 introduced multiple entity loading and rendering so that tasks such as fetching field values from the database could be done once for all nodes with an IN() query rather than executed individually:

<?php

function **render\_entities**($ids) {

$entities = entity\_load\_multiple($ids);

**return = entity\_view\_multiple($entities);**

}

?>

By using the multiple load and view functions, assuming 10 nodes need to be loaded and rendered, 10 similar queries to the same table can be reduced to just one. Since an individual node load could require 10 or 20 database queries, this can result in dozens or hundreds of database queries saved when loading and rendering multiple nodes at the same time.

Note that this applies to hook implementations as well; for example, hook\_entity\_load() acts on an array of entities.

One often overlooked hook is hook\_entity\_prepare\_view(). Often, custom themes will need to add fields from user accounts/profiles when rendering nodes or comments—this could be the user’s full name, avatar, registration date, etc. A common pattern for this is preprocess. Let’s take nodes as an example:

<?php

template\_preprocess\_node(&$variables) {

$node = $variables['node'];

$variables['account'] = user\_load($node->uid);

*// Set up custom variables based on account here.*

}

?>

When rendering several different nodes or comments by different authors, this pattern can result in a lot of round trips to the database as each account is fetched individually. The following example provides the same functionality while resolving the performance issue:

<?php

hook\_entity\_prepare\_view($entity\_type, $entities) {

if ($entity\_type != 'node') {

return;

}

$uids = array();

foreach ($entities as $entity) {

$uids[] = $entity->uid;

}

$accounts = user\_load\_multiple($uids);

foreach ($entities as $entity) {

$entity->account = $accounts[$entity->uid];

}

}

?>

Then $entity->account is available in preprocess:

<?php

template\_preprocess\_node(&$variables) {

$account = $variables['node']->account;

}

?>

**Caching**

Caching is often the quickest way to solve a performance issue. By adding caching in a particular code path, you can ensure that it will only be executed on cache misses.

Before adding caching, though, there are a few things to consider:

* Is it possible to optimize the code so that it doesn’t need to be cached?
* Is there already caching of the code at a higher level, for example page caching, that might affect the hit rate?
* Will the cached code path be considerably quicker than the current code path?
* Does the cache need to be cleared on particular events? Is it OK for it to be stale sometimes?
* Is the code run multiple times with the same output during a single request?

**Static Caching**

When code is run multiple times per request, a common optimization is to add a static cache around it. For example, you might rewrite the following code:

<?php

function **my\_function**() {

return something\_expensive();

}

as

<?php

function **my\_function**() {

static $foo;

if (!isset($foo)) {

$foo = something\_expensive();

}

return $foo;

}

?>

**Because $foo is declared as static, it will be held in memory for the duration of the request regardless of how many times the function gets called**. This means once this function has run once, it will run the isset() check and then immediately return.

While it only takes a couple of lines of code to add a static cache, doing so has implications that aren’t always immediately obvious.

Let’s look at the code inside something\_expensive():

<?php

function **something\_expensive**() {

return friends\_count($GLOBALS['user']);

}

?>

Whoops. If $GLOBALS[‘user’] changes during the request, then something\_expensive() will return different output. This often happens during automated tests using Drupal’s *simpletest* adaption, or in a *drush* process that might be sending emails to multiple different users.

It’s not impossible to fix this, of course. For example, we can key the cache based on the global user’s ID:

<?php

function **my\_function**() {

static $foo;

global $user;

if (!isset($foo[$user->uid])) {

$foo[$user->uid] = something\_expensive();

}

return $foo[$user->uid];

}

?>

Now, regardless of how many times the global user object is swapped out during the request, our function will return correctly, whilst still statically caching the results.

But the problems don’t end there. What if the number of friends the user has changes during the request as well? This might well happen during a functional test or a long-running *drush* job. Additionally, this is where memory usage starts to be a problem: a *drush* job processing one million users could eventually end up with a million items in this static cache.

**Drupal core has a solution for this in the form of the drupal\_static() functio**n. This operates similarly to static caching, except that the static cache can be accessed from different functions, both for retrieval and for reset.

Now our function looks like this:

<?php

function **my\_function**() {

*// Only this line changes.*

$foo = &drupal\_static(\_\_FUNCTION\_\_);

global $user;

if (!isset($foo[$user->uid])) {

$foo[$user->uid] = something\_expensive();

}

return $foo[$user->uid];

}

?>

Code in unit tests that updates the user’s friends count or needs to reclaim some PHP memory can then call **drupal\_static\_reset(‘my\_function’)**to empty the static cache.

Since drupal\_static() is a function call, it has a lot more overhead than declaring static and including an isset() check. This can lead to a situation where static caching is added to micro-optimize a function, then converted to drupal\_static() for testing purposes, which leads to the function being slower than when it had no caching at all. If you absolutely need to use drupal\_static() and your function is going to be called dozens or hundreds of times during a request, there’s the drupal\_static\_fast pattern:

<?php

function **my\_function**() {

static $drupal\_static\_fast;

if (!isset($drupal\_static\_fast)) {

$drupal\_static\_fast['foo'] = &drupal\_static(\_\_FUNCTION\_\_);

}

$foo = $drupal\_static\_fast['foo'];

global $user;

if (!isset($foo[$user->uid])) {

$foo[$user->uid] = something\_expensive();

}

return $foo[$user->uid];

}

?>

This adds testability and performance at the expense of quite a bit of complexity.

There are two issues with my\_function() now. One is a development process issue, and the other is architectural.

In terms of process, if we look back at the original function, we can see it’s only a wrapper around something\_expensive(). While a real example probably wouldn’t be a one-line wrapper, if the only thing that needs caching is something\_expensive(), this isn’t the right place to add that caching. What we should have done was add the caching directly to something\_expensive(), which also knows about any global state it depends on and any other factors that might influence the result (and, if you’re lucky, is in a contributed module rather than your custom code).

When you add caching to a wrapper rather than to the function itself, the following bad things happen:

* Any other code that calls the function (here, something\_expensive()) does not get the benefit of the static caching.
* If the function or another function that calls it adds static caching at a later point, the same data will be added to the cache twice, leading to both higher memory usage and potentially hard-to-find cache invalidation bugs.

From an architectural/readability perspective, we can see the gradual change from a very simple function to one that is balancing various variables in global state. A major change in Drupal 8 has been the migration from procedural APIs to object-oriented code based on dependency injection. Most classes are loaded via a factory method or plug-in manager, or accessed from the dependency injection container. When this is the case, simply using class properties is sufficient for managing state between methods, and no static caching is necessary at all.

**Persistent Caching**

Drupal core ships with a rich caching API, defaulting to database caching but with contributed support for files, Memcache, Redis, MongoDB, APC, and other backends.

While static caching allows code to skip execution when called within a single PHP request, persistent caching is shared between PHP processes and can retain data for anything from a few seconds to several weeks.

The Cache interface is well documented on the [Drupal API site](https://api.drupal.org/api/drupal/core!lib!Drupal!Core!Cache!CacheBackendInterface.php/interface/CacheBackendInterface/8), and there are numerous examples of basic usage in core modules. Rather than duplicating that information here, we’ll discuss some of the lesser known features and ones new to Drupal 8.

**Cache chains**

A new feature in Drupal 8 is the *cache chain backend*, a means of stringing together different cache storage backends in a way that is transparent to the calling code. This feature is primarily designed for combining two persistent storage backends together—for example, APC and database caching—in order to get the best of both. With an APC and database chain, the cache will check APC first and return immediately if an item is found. If not, it will check the database and then write back to APC if the item is found there; and on cache misses, it will write to both. It’s also possible to use the memory backend shipped with Drupal core and any other persistent backend to emulate the static + persistent caching pattern shown earlier, without the code complexity.

**Cache bins**

Drupal core defines several different cache bins, including "bootstrap" for information required on every request, the default cache bin, and use-case-specific bins such as "page" which is only used for cached HTML pages. The cache API not only allows for storage to be swapped out but also allows it to be changed for each cache bin, via the $conf[*cache\_backends*] variable in Drupal 7 and the dependency injection container in Drupal 8.

The bootstrap cache bin is designed for items needed for every request; it’s used primarily by low-level Drupal APIs such as the theme registry or hook system. The cache items in this bin tend to be invalidated infrequently—often when a module is enabled or disabled—and since they’re requested all the time will have an extremely high hit rate.

On the other hand, the "block" cache bin is used to cache the output of Drupal’s blocks system. Blocks may have different cache items per role, per user, and/or per page, which can result in hundreds of thousands or more potential entries in the bin. The bin is also cleared often on content updates, so it has a high insert/delete/update rate.

In most cases, sites will want to set up a single cache backend such as Memcache or Redis to handle all cache bins, but the option is there to use different backends with different bins if desired.

When using the cache API, you’ll likely use the default cache bin, or create a custom bin. A custom bin should only be used if there’s going to be a very large amount of data to cache.

**getMultiple()/setMultiple()/deleteMultiple()**

As with entity loading, the cache API allows for loading, setting, and deleting multiple cache objects at once. Any situation where you know the cache IDs of multiple objects in advance is a candidate for using these methods, and many different storage backends natively support multiple get, allowing a single round trip to the cache storage and a shorter code execution path.

**Cache tags**

A new feature of the core cache API in Drupal 8 is cache tags.

There is often confusion between cache tags as a concept and cache IDs, so let’s explain cache IDs first.

When creating a cache ID in Drupal, the following conventions are important:

* Use the module name or another unique prefix at the start of the cache ID to avoid naming conflicts with others.
* Where a cache ID depends on context, include enough information about this context in the ID to ensure uniqueness. That is, for a cache item that varies per user, you might use:

<?php $cid = 'my\_module:' . $uid; ?>

If it varies by language as well, then use:

<?php $cid = 'my\_module:' . $uid . ':' $langcode; ?>

In this case, the semantics of what makes up the cache ID aren’t important; all that matters is that one user doesn’t get presented content that was cached for another user or translated in a different language to the one they’re viewing the content in..

One exception to this is *key-based invalidation*—using the updated timestamp of an entity as part of the cache key means that when the entity is updated, so is the cache key, resulting in a cache miss and new cache entry without having to explicitly clear the old key.

Cache tags, rather than guaranteeing the uniqueness of cache items, are intended for cache invalidation.

A good example of this is entity rendering. Entities may be rendered on their own with multiple view modes, as part of a listing of multiple entities via Views, as part of a block, or embedded within the rendering of another entity via entity references.

A rendered node may include information from referenced entities, such as the name of the user that authored the node and that user’s avatar. A Views listing might include multiple nodes like this.

To maintain coherency when entities are updated, there are two common approaches:

*Set long TTLs and clear all caches of rendered content*

The cache will be completely emptied whenever a single item of content is updated, even though the majority of the cache will be unaffected. On sites with frequent content updates, this approach can lead to low hit rates and the potential for cache stampedes. However, the cache will always be accurate.

*Set short TTLs so that content is only stale for a few seconds or minutes*

This results in lower hit rates regardless of the frequency of content updates. However, not explicitly clearing the cache all at once when an item is updated means there’s less likelihood of cache stampedes.

Cache tags allow for a "best of both worlds" scenario, where all cache items that include an entity are tagged with that entity’s ID, and saving the entity invalidates those cache items but no others. This allows for both cache coherency (assuming consistent tagging in the first place) and longer TTLs.

**CacheArray**

CacheArray was originally added to Drupal 8 but has been backported to Drupal 7, along with several patches integrating it with core subsystems. As a highly dynamic system, and with so much functionality provided by modules, Drupal has evolved to carry a lot of metadata about what functionality is provided from where. This includes the theme registry (a large array of all theme hooks, templates, and preprocessors), the schema cache (which contains metadata about every database table defined by a module, which often reaches 200 or so), and several other registries. On a default install of Drupal core, these usually reach a few hundred kilobytes at most; however, many Drupal sites end up with as many as a hundred or even several hundred contributed modules enabled, each of which may be defining new database tables, theme templates, and the like.

Prior to Drupal 7.7, each subsystem would store these arrays in one large cache item. This meant that for the theme registry, every theme function or template registered on a particular site would be loaded on every page—including theme functions for specific administrative tables that might not be used, or for functionality that might not be exposed on the site itself due to configuration. For the schema cache, while the schema metadata is only used for tables passed to drupal\_write\_record() or drupal\_schema\_fields\_sql()—often as few as 10–15 tables on most sites—metadata about every database table on the site would nevertheless be loaded from the cache for every request.

[CacheArray](http://api.drupal.org/api/drupal/core!lib!Drupal!Core!Utility!CacheArray.php/class/CacheArray/8) provides a mechanism to drastically reduce the size of these cache entries by emulating a PHP array using ArrayAccess. When an array key is requested that hasn’t already been cached, it’s treated as a cache miss and looked up, and then the array is populated with the returned value. At the end of the request, any newly found array keys and values get written back to the cache entry so that they’ll be a cache hit for the next request. This allows the cache item to be built on demand, populated only with data that is actually in use on the site and often excluding infrequently accessed items such as those used for administrative pages that may not be visited during normal site operation. Relatively few contributed modules need to maintain as much metadata as some of these core subsystems, but CacheArray provides a solution to this problem when you run into it.

CacheArray is in the process of being replaced by [CacheCollector](https://api.drupal.org/api/drupal/core!lib!Drupal!Core!Cache!CacheCollector.php/class/CacheCollector/8" \t "_top) in Drupal 8. CacheCollector has the same internal logic but uses public methods for get and set instead of ArrayAccess.

**Render caching**

Drupal’s render API takes a structured array of data and converts it to HTML, running it through the theme system and collecting associated assets such as CSS and JavaScript. One of the more powerful but underused features of the render system is its integrated cache handling.

When writing code that generates HTML, there are two main phases that the content goes through:

* Building the array of data (e.g., a list of nodes based on the results of a query)
* Rendering the array to HTML, which mainly involves running it through the theme system

Render caching allows the majority of time spent in these operations to be skipped. We’ll take the example of a custom block that shows the five most recently published article titles, taking it from no caching at all to using the render cache as much as possible:

*/\*\**

*\* Implements hook\_block\_info().*

*\*/*

function **example\_block\_info**() {

$blocks['example\_render\_cache'] = array(

'info' => t('Render caching example.'),

'cache' => DRUPAL\_CACHE\_CUSTOM,

);

return $blocks;

}

*/\*\**

*\* Implements hook\_block\_view().*

*\*/*

function **example\_block\_view**($delta = '') {

switch ($delta) {

case 'example\_render\_cache':

$query = new EntityFieldQuery();

$query->entityCondition('entity\_type', 'node')

->entityCondition('bundle', 'article')

->propertyCondition('status', 1)

->range(0, 5)

->propertyOrderBy('created', 'ASC');

$result = $query->execute();

$nids = array\_keys($result['node']);

$nodes = node\_load\_multiple($nids);

$titles = array();

foreach ($nodes as $node) {

$titles[] = l($node->title, 'node/' . $node->nid);

}

$block['subject'] = t('Render caching example');

$block['content'] = array(

'#theme' => 'item\_list',

'#items' => $titles,

);

break;

}

return $block;

}

When the block is rendered with each request, first the hook\_block\_view() implementation is called. Then the resulting render array is run through drupal\_render() (the second phase).

Just adding #cache to the render array would skip theming, but the entity query and loading would continue to happen with every request without some reorganization. Render caching allows us to skip that work as well, by moving that code to a #pre\_render callback. This is the most complicated aspect of using render caching, so rather than adding the cache first, we’ll start by moving that code around.

hook\_block\_view() now looks like this:

*/\*\**

*\* Implements hook\_block\_view().*

*\*/*

function **example\_block\_view**($delta = '') {

switch ($delta) {

case 'example\_render\_cache':

$block['subject'] = t('Render caching example');

$block['content'] = array(

'#theme' => 'item\_list',

'#pre\_render' => array('\_example\_render\_cache\_block\_pre\_render'),

);

break;

}

return $block;

}

*/\*\**

*\* Pre-render callback for example\_render\_cache block.*

*\*/*

function **\_example\_render\_cache\_block\_pre\_render**($element) {

$query = new EntityFieldQuery();

$query->entityCondition('entity\_type', 'node')

->entityCondition('bundle', 'article')

->propertyCondition('status', 1)

->range(0, 5)

->propertyOrderBy('created', 'ASC');

$result = $query->execute();

$nids = array\_keys($result['node']);

$nodes = node\_load\_multiple($nids);

$items = array();

foreach ($nodes as $node) {

$items[] = l($node->title, 'node/' . $node->nid);

}

$element['#items'] = $items;

return $element;

}

hook\_block\_view() now returns only the minimum metadata needed; the bulk of the work is transferred to the render callback, which will be called by drupal\_render() itself when the element is rendered.

Once this is done, adding caching requires only a small change to hook\_block\_view():

*/\*\**

*\* Implements hook\_block\_view().*

*\*/*

function **example\_block\_view**($delta = '') {

switch ($delta) {

case 'example\_render\_cache':

$block['subject'] = t('Render caching example');

$block['content'] = array(

'#theme' => 'item\_list',

'#pre\_render' => array('\_example\_render\_cache\_block\_pre\_render'),

'#cache' => array(

'keys' => array('example\_render\_cache'),

),

);

break;

}

return $block;

}

Adding #cache means that drupal\_render() will check for a cache item before doing any other processing of the render array, *including* the #pre\_render callback. Profiling a page with this block before and after should show that the EntityFieldQuery and node loading has been removed on cache hits. See [???](http://chimera.labs.oreilly.com/books/1230000000845/ch01.html) for more information about how to check this.

**Queues and Workers**

Drupal core ships with a robust queue API, defaulting to MySQL but with contributed projects providing support for Redis, Beanstalkd, and others.

The queue API is most useful when you have expensive operations triggered by actions on the site. For example, saving a node or comment may require updating the search index, sending email notifications to multiple recipients, and clearing various caches. Performing all of these actions directly in hook\_node\_update() will mean the request that actually saves the node takes considerably longer, and introduces single points of failure in the critical path of updating content. Depending on the implementation, failures in search indexing or sending emails may show up as errors to the end user or interrupt the content saving process altogether.

Instead of doing all this work inline, in your hook\_node\_update() implementation, you can create a queue item for that node; then, in the worker callback, you can perform whichever tasks on it are necessary.

This has the following advantages:

* Expensive processing is taken out of the critical path of saving nodes into a background process. This allows the Apache process to be freed up quicker and pages to be served more quickly to users.
* The background process may be run by *drush* or a queue daemon. Any operations that require high memory limits won’t bloat Apache, and they don’t necessarily need to run on the main web server at all. If queues are processed by Jenkins, it’s also possible to isolate reporting of failures for particular queues.
* Multiple queue workers may run at the same time, allowing infrastructure usage to be maximized when there are lots of items in various queues. In contrast, Drupal’s hook\_cron()only allows one cron invocation to run at a time.
* Queue items are processed individually and can be returned to the queue if not successful. For example, if a queue item needs to call an external service but the API call fails with a 503 response, it can be returned to the queue to be retried later.

In sum, pages can be served to end users faster, you have more flexibility when scaling up your infrastructure, and your application will be more robust against failures or performance issues in external providers.

**Cache Stampedes and Race Conditions**

As sites reach large numbers of simultaneous processes, the potential for stampedes and race conditions increases.

A stampede can happen when a cache item is empty or invalid and multiple processes attempt to populate it at the same time. Here’s an example with Drupal 7’s variable cache:

* Process A requests the variable cache, but there is no valid entry, so it starts loading the variables from the database and unserializing them.
* Process B then comes in; there is no cache entry yet, so it also queries the variables from the database.
* Process C comes in and does the same thing.
* Process A finishes building the page and caches it.
* Process D requests the page and gets the cached item.
* Processes B and C finish and overwrite the cache item with their own identical versions.

In this case, only one cache item was needed, but it was created three times.

If this is an expensive task, it can put the server under high load as multiple different processes all do duplicate work.

There are two approaches to handling this scenario:

* When it’s OK for a few requests to be served an invalid cache item, it’s possible to use the $allow\_invalid parameter to $cache->get() so that invalidated but still present cache items are returned by the API. The first request to get an invalid cache item can acquire a lock using Drupal core’s lock API, then proceed to build the new cache item and return it to the caller. Subsequent requests will fail to acquire the lock and can immediately return the stale cache item; this will happen until the new cache item is available.
* When the cache item must be up to date at all times, or if a cache item is completely empty, it’s not possible to serve stale content. The first process to get a cache miss will still acquire a lock and proceed to build the fresh cache item. If the item is very expensive to build, then subsequent requests can be put into a holding pattern using $lock->wait(). This will return as soon as the lock is released, after which the cache should be available.

Using the locking system can have its own problems—when a cache stampede turns into a lock stampede—and it would be remiss not to discuss these:

* By default, acquiring a lock requires a database write, and polling for locks queries the database quite frequently. Locking to save one inexpensive database query can be counterproductive in terms of performance since it may have as much overhead as rebuilding the cache item. The lock API has pluggable storage, so this can be improved by installing one of the alternative backends.
* Items that are cached per page are less likely to be requested simultaneously than items cached once for the whole site. Where there is very little chance of a cache stampede, the extra overhead of acquiring the lock is not worth it for these items.
* Items that are invalidated very frequently—say, every 10 seconds—will result in a constant acquiring and freeing of locks. Since processes that don’t acquire locks usually poll to see if the item is created, they may miss the window of the valid cache item and continue to poll.

If you are running into issues with locking, consider whether the lock may be making things worse rather than better. Alternatively, it may be necessary to rethink the functionality altogether; for example, refactoring a per-page element to work across all pages on the site.

# Writing efficient code

### Try to combine database queries instead of requesting in a loop

You should try to avoid queries in loops. Instead, build bigger queries and send them at combined. This applies as well for INSERT INTO or DELETE FROM queries with multiple rows.

### Take care with getters and setters

The direct access is more then twice performant as the encapsulation. It's a shame, but it means you have to check where you need proper encapsulation and where not, or better said where you have a lot of accesses.

### Cache query results

**In case** you have to run a query in a loop, you can cache your results in a temporary array and put a check before query execution that you do not have result for the query iteration in your temporary array

# Drupal-based site caching

The core caches in Drupal are the page and block caches. In addition, you can use Drupal modules to cache views, panels, and entities, or to cache pages for authenticated users. You can also implement caching in custom code, using the Drupal Cache APIs.

## Page caching

The main Drupal cache is a page cache; it caches complete pages. This means that the page cache has limited value if your website (like many) personalizes each page for signed-in visitors. Even a simple personalized welcome message on a page means that the identical page is unlikely to exist in the cache and must be regenerated for each page request.

By default, Drupal doesn't enable page caching; without it, every time a user visits a page, Drupal rebuilds the page. Page caching isn't in effect for users logged in to Drupal, but you should enable page caching for any website that receives any anonymous traffic.

### Drupal 7 cache settings

On the Performance page at **Configuration > Performance**:

1. Select **Cache pages for anonymous users**. And **Cache Block**
2. Set the **Minimum cache lifetime**. The minimum cache lifetime prevents Drupal from clearing page and block caches after changes are made to nodes or blocks for a set period of time. This can cause unexpected behavior when editing content or when an external cache such as Varnish is employed. Therefore, minimum cache lifetime should be used with caution. If you are unsure, leave the minimum cache lifetime set to 0.
3. Set the **Expiration of cached pages**. This is the maximum time a page can stay in the Varnish cache. If you set this to <none>, the max-age header Drupal sends to Varnish is 0, which means it will not be cached. A reasonable setting is 6-12 hours.

Block caching

By default, Drupal doesn't cache individual blocks. While anonymous users will see cached blocks if you have page caching enabled, you should enable block caching on all websites to speed up page browsing for logged-in users.

1. Sign in to your website using an account with administrator privileges.
2. Go to the Performance page at **Configuration > Performance** ([site path](https://docs.acquia.com/articles/url-paths-different-drupal-versions)).
3. Select the **Cache blocks**check box.
4. Click **Save configuration**.

If your website has some blocks that prevent you from enabling block caching, consider the [AJAX Blocks](https://www.drupal.org/project/ajaxblocks)module.

The [Views](https://www.drupal.org/project/views)module is a query builder; any view that it creates queries your database one or more times. This can result in multiple database queries for any given page load. By default, Views doesn't cache any of its queries or data. While Drupal's page caching will mask some of the problem, websites with a large amount of traffic coming from authenticated users will see heavier load unless you cache the views.

Setting up view caching

You can start caching your views by setting up time-based caching on any view that will be seen regularly.

To set up caching for a view, complete the following steps:

1. Sign in to your website, and then go to **Structure > Views** (Drupal 7) or **Site building > Views** (Drupal 6).
2. Find the view that you want to configure, and then click its **Edit** link.
3. In the **Advanced** settings section, click None next to **Caching**.
4. Select **Time-based**, which gives you the opportunity to cache query results or rendered output.
5. Click **Apply** to save the new caching value.
6. Click **Save** to save the edits to your view.

You need to consider settings individually per view. Even a five-minute cache can make a significant difference in website performance. A blog post on [Optimizing Drupal Views the Right Way](https://www.silviogutierrez.com/blog/optimizing-drupal-views-right-way/)offers several suggestions for Views caching.

# Optimizing Drupal Views the Right Way

The Views module is one of the most impressive features Drupal offers. With some clicks here and there, you can build a rich display of data in minutes. But to offer this kind of flexibility, Views makes some assumptions that can hamper your site's performance. In this post, you'll find out how to improve performance dramatically.

First, a quick technical explanation of what happens. Drupal shares the main record for each content type in the node table. That means that regardless of whether an entry is a blog post or an event, it will get a record in this table. This has the advantage that building aggregated lists of all entries is quite easy.

**On Joins**

How does this tie into Views? When you build a view that uses a field specific to a content type - for example, event date - it doesn't know if every entry will have this field or not. So when querying the database, it tells it "give me this node. And if it has a corresponding field called event date, give me that as well." This is called a LEFT JOIN, and it's quite slow. If on the other hand, we changed the query ever so slightly to "give me this node and the corresponding field event date," then the whole thing will execute much quicker. By doing this, the query is executing an INNER JOIN which joins each node entry with each event entry one to one. Keep in mind, this will only work if you want the result set to contain events only. If you need to mix events and blog posts, you're out of luck.

**On Pagers**

Guess what the most expensive part of your view is? That's right, the pager at the bottom. Turns out, counting in databases is slow. Really slow2. So get rid of as many COUNT queries as you can. What's this got to do with pagination? To accurately paginate, Views executes a COUNT query on the entire result set. That means that even if you're displaying 20 records out of a million, Views will need to count all 20 million rows. This takes a long time.

So what are your options? You've got two. One is to get rid of pagination entirely. The other is to use a less accurate pager. We won't go over the former, because clients/product managers can be notoriously inflexible - good luck trying to convince them to use no pagination. But a less accurate pager? They can probably live with that.

So how does a less accurate pager work? First, it eliminates the "last" option. That means you don't need to know how many total results there are. Skipping more than 2 pages ahead at a time is also nixed. With these compromises, you can get rid of the COUNT query and simply use the current offset to build the pager. You'll only be able to accurately jump to pages before the one you're on, but often this is good enough.

**On Distinct**

DISTINCT clauses have their time and place - never. No but really, try to avoid them. Most of the time, you can eliminate the need for them by crafting a better query. Duplicates rows are usually a symptom of a bigger problem. So you shouldn't mark your view queries as needing DISTINCT unless you're absolutely certain it's needed. Or if you're lazy. Regardless of the reason, it will slow down your query. Worse - when you use DISTINCT in a view, a GROUP BY clause will be inserted as well. This will also slow down your query significantly. Fortunately, disabling DISTINCT will rid you of both problems. Of course, do check to make sure you're not getting duplicate results. If you're using INNER JOIN as described above, you most likely won't.

**The Changes**

After all that technical talk, let's actually apply these changes. Three assumptions:

Let's assume your view's machine name is my\_view.

Let's also assume it's joining to a table called content\_type\_event for the additional event information.

Let's pretend you've got a module floating around called my\_module.

Here's how to do this The Right Way

|  |  |
| --- | --- |
| 2  3  4  5  6  7  8  9  10  11  12  13 | <?php  */\*\**  *\* Implements hook\_views\_query\_alter().*  *\**  *\* Disables distinct and turns LEFT joins into INNER joins. This drammatically*  *\* speeds up performance.*  *\*/*  **function** my\_module\_views\_query\_alter(&$view, &$query) {  **if** ($view->name == "my\_view") {  $query->table\_queue['content\_type\_notice']['join']->type = 'INNER';  $query->distinct = 0;  }  } |

What about the pager? That's even easier. Just download the lite pager module and select it as the paging option for your view.

# Caching to improve performance

### Opcode Caching (PHP - high CPU)

PHP out of the box is a dynamic language and can lead to heavy CPU usage on web servers. There are multiple types of opcode caching add-ons for PHP available that will convert your .php page into memory (byte code) to provide a major benefit in load time and reduced CPU usage. Each of these will require root access to install and have their own specific configurations that will need some attending before using. Once configured and enabled they can provide a substantial benefit to a slow site and greatly reduce CPU usage from PHP.

**APC and the APC project page**

### Database Caching

Database caching can be provided by a distributed memory object caching system, such as **Memcache**. Memcached allows you to allocate memory where you have more than you need and make it accessible to areas where you have less than you need. When implemented with Drupal, Memcached can store the result of your database query’s in the memory for a specified time, reducing database traffic.

### Web Server (Proxy) Caching

HTTP acceleration, or Web Server (Proxy) Caching, will significantly reduce resource requirements and page load times. **Varnish** Cache, is a widely-used HTTP accelerator for Drupal sites.

HTTP acceleration is handled by a reverse proxy. A reverse proxy is a type of proxy server that retrieves resources (pages, images, files, etc.) prior to being requested by a website visitor. These resources are stored in virtual memory and are quickly retrieved by the proxy server when requested. Reverse proxies are heavily threaded and optimized for data retrieval.

### Authenticated Users

Performance Modules

If better optimization is required, you might install one or two performance modules. For example, if the site is on a shared server, try Boost (static page caching for non-logged in visitors), complemented with Authcache -in CR's file (for logged in members).

If on a VPS or dedicated server, there are several suitable options: Boost (for example on a low RAM VPS), Authcache, Cache Router (which includes , Memcache...), Varnish, etc. Memcache (or Memcache Storage) and Varnish are also especially suitable when load-balancing multiple servers.

# Changing PHP memory limits

It's important to understand that increasing the amount of memory *each* PHP process can use decreases the number of *concurrent* processes that can run. Concurrent processes are key to the maximum number of page requests you can serve at any given time.

For example, lets assume a server with 2 GB of memory free for PHP after accounting for other processes running on it (the OS, Apache, and so on).

|  |  |  |  |
| --- | --- | --- | --- |
| **PHP memory limit** | 128MB | 256MB | 512MB |
| **Max number of concurrent processes** | 16 | 8 | 4 |

*Memory limit/concurrency relationship with 2GB RAM*

## How to change memory limits

### php.ini

memory\_limit = 64M

### php.ini in the Drupal root folder

memory\_limit = 64M

### . htaccess

### php\_value memory\_limit 64M

### settings.php

ini\_set('memory\_limit', '64M');

# Content Delivery Network [CDN]

CDNs are globally distributed networks of proxy servers. A basic CDN integration will offload static assets (like images, videos, and sometimes CSS and JS) to the CDN. More advanced CDNs can deliver entire pages.

CDNs offer several advantages over serving all traffic directly:

* Assets can be cached in a proxy which is geographically closer to the end user, which usually leads to lower latency and increased download speed;
* Each page response is shared between your origin server and the CDN, meaning that your origin server can serve more concurrent requests;
* By splitting a response across multiple servers, enhanced HTTP performance can be achieved via domain sharding, although this is becoming less advantageous with the increasing adoption of HTTP/2 and SPDY;
* Some CDNs offer page optimisation services which can further enhance performance for end users.

# Design for Low Bandwidth

Drupal offers a variety of tools and features to reduce bandwidth requirements, including:

### Adaptive Images

Your site is being increasingly viewed on smaller, slower, low bandwidth devices. On those devices your desktop-centric images load slowly, cause interface lag, and cost you and your visitors un-necessary bandwidth and money. Drupal includes the functionality to deliver small images to small devices; reducing bandwidth and resource requirements.

Drupal is capable of detecting a visitor's screen size and automatically create, caches, and deliver device appropriate re-scaled versions of your web page's embedded HTML images. No mark-up changes are required. It is intended for use with responsive designs. See [Adaptive Image](https://www.drupal.org/project/adaptive_image) module for more information.

### Aggregated and Compressed Stylesheets and JavaScript

Drupal has many stylesheets and JavaScript files. Of course, you want to keep the number of files at a minimum on your live sites for performance reasons, so Drupal includes options to aggregate and compress CSS and JavaScript files.

Drupal aggregates files in two ways: it creates a per-site aggregation file from files that would be loaded on every page, and it creates a per-page aggregation files for the remaining files that are conditionally loaded depending on the page. For CSS files, it further aggregates by media type.

Enabling aggregation and compression for CSS files on all live sites is highly recommended, as it will lower bandwidth and resource requirements for website visitors. Aggregation will also minimize the number of HTTP requests required to load a page; each request has an amount of overhead and latency associated with it. Too many requests would add delays to page loading.

### Support for Browser Caching

A user’s browser can keep a static file (image, stylesheets, etc.) without checking the server for an update and downloading it again. Therefore, each time someone visits a page on your website, the visitor will not be required to download the static content again, reducing bandwidth, resource requirements, and page load times. For more information regarding browser caching, see [mod\_expires](http://httpd.apache.org/docs/2.2/mod/mod_expires.html) documentation.

# Increase upload size in your php.ini

**Drupal's limits on upload file size are determined by your server's PHP settings (as well as Drupal specified settings that can be set at Admin > Site Configuration > File Upload). The default values for PHP will restrict you to a maximum 2 MB upload file size.**

On the settings page for the upload module, Drupal calculates and displays the maximum file size that you can set based upon two PHP settings: 'post\_max\_size' and 'upload\_max\_filesize'. Since 'post\_max\_size' is the limit for all the content of your post, many people choose 'post\_max\_size' to be a multiple of 'upload\_max\_filesize' to allow multiple files to be uploaded, but this is not essential. The upload module limits the size of a single attachment to be less than either post\_max\_size, or upload\_max\_filesize, whichever is smaller. The default PHP values are 2 MB for upload\_max\_filesize, and 8 MB for post\_max\_size.

Depending on your host, changing these two PHP variables can be done in a number of places with the most likely being php.ini or .htaccess (depending on your hosting situation).

For example, to increase the limit on uploaded files to 10 MB:

* Add the below to the relevant php.ini file (recommended, if you have access). Note that for some hosts this is a system-wide setting. However, for hosts running PHP as a CGI script with suexec (for example) you may be able to put these directives in a **php.ini** file in your Drupal root directory.
  + upload\_max\_filesize = 10M
  + post\_max\_size = 10M
* Add the below to your .htaccess file in your Drupal root directory.
  + php\_value upload\_max\_filesize 10M
  + php\_value post\_max\_size 10M

# Optimizing Drupal to load faster (Server, MySQL, caching, theming, HTML)

Basic settings

1. Configure cron job (for drupal 6 <http://drupal.org/project/poormanscron> )
2. Make sure all cache tables are clearing properly especially cache\_form
3. Enable cache options in performance page

### Theme optimization

1. Manually Remove blankspaces and comments from .tpl
2. No indentation in .tpl
3. Turn on CSS and JS aggregation in the performance page
4. Manually reduce css file size by removing duplicate and combine similar together
5. Move codes to functions which should be in a custom common module. Use functions for similar problems instead of coding separately. Refer core API

### Coding standard and proper use of already existing core API

1. <http://drupal.org/coding-standards>

Secure codes

1. <http://drupal.org/writing-secure-code>

DB Query optimization in codes

1. Join db queries whenever possible
2. For Db update and insert, use core API
3. Use drupal standard <http://drupal.org/coding-standards>

DB table optimization

1. <http://drupal.org/project/db_maintenance>

Disable unnecessary modules

1. Devel
2. Statistics
3. Update status
4. Use syslog instead of Database logging

Remove unnecessary contents and others

### Cache modules

1. Make use of Memcache module to reduce Database overhead (<http://drupal.org/project/memcache/> ) Or
2. APC (for drupal 7, <http://drupal.org/project/apc> )  
   (for drupal 6, <http://drupal.org/project/cacherouter> + <http://drupal.org/project/boost> (optional) )
3. <http://drupal.org/project/authcache>
4. Some module may help improve  
   <http://drupal.org/project/ajaxify_regions> (or <http://drupal.org/project/ajaxblocks> )

### Make changes according to Google Pagespeed and yahoo YSlow suggestions

MySQL Settings

1. Cache Size say 32MB in MySQL

Apache settings

1. DNS lookup : OFF
2. Set FollowSymLinks everywhere and never set SymLinksIfOwnerMatch
3. Avoid content negotiation. Or use type-map files rather than Options MultiViews directive
4. KeepAlive on, and KeepAliveTimeout very low (1 or 2 sec)
5. Disable or comment access.log settings
6. Enable mod\_deflate or mod\_gzip
7. Install APC server with higher memory limit apc.shm\_size = 64

### Turn Page Caching On

### Turn Views caching on

# Optimizing MySQL

* Start with an appropriate MySQL [option file](http://dev.mysql.com/doc/refman/5.0/en/option-files.html). For servers with at least 2GB RAM dedicated to MySQL we recommend my-huge.cnf. For servers with a lot of writes, we recommend my-innodb.cnf instead of the default MyISAM engine type.
* If a query is called at least twice with no modifications to the queried tables a significant performance improvement can be gained by avoiding the processing of the query and the execution of the query by reading the query from the [MySQL query cache](http://dev.mysql.com/doc/refman/5.1/en/query-cache.html). To learn how to set up the query cache read [a practical set-up](http://dev.mysql.com/doc/refman/5.1/en/query-cache.html).
* Be sure to have enough cached threads or you will launch too many new threads as described in [this story about a Yahoo site](http://jeremy.zawodny.com/blog/archives/000173.html).
* The biggest performance boosts can come from identifying and tuning the slowest queries using the [MySQL slow query logs](https://dev.mysql.com/doc/refman/5.0/en/slow-query-log.html).
* You can use the [DB Maintenance](https://www.drupal.org/node/41588)module or use the mysqlcheck commands below in a cronjob.
* MySQL supports many different engine types including Archive (MySQL 5+), MyISAM, and InnoDB. Performance sites should use InnoDB for most tables particularly tables such as node that get a lot of writes and reads. MyISAM exclusive table locks for updates before selects versus InnoDB's row level locking can mean MyISAM blocks reads if there are many writes. Convert from MyISAM to InnoDB.
* In MySQL 5 a new table type called the archive table type was introduced to deal with common requirements for web applications like access logs where only INSERTS and SELECTS were done. If tables such as the Access Log table are determined to not have DELETE, UPDATE, and REPLACE then this MySQL engine type can often offer significant performance improvements as they have done for sites like Slashdot, Yahoo, and Live Journal.
* Make use of the Percona Toolkit, especially pt-variable-advisor to check for problem settings and pt-mysql-summary to get an at-a-glance look at your configuration.
* Avoid RAID-5 in a high write environment. It can degrade performance significantly.
* Consider changing the IO scheduler if you are running on a dedicated MySQL server. The noop or deadline schedulers can have significant performance increases over anticipatory and cfq.

# Server tuning considerations

### Identifying Drupal performance goals

### Analyzing your site's traffic and resource consumption

### Understanding and configuring your stack for performance

### Understanding LAMP performance. This [LAMP performance study](http://web.archive.org/web/20061115173834/http:/sourcelabs.com/pdfs/SourceLabsApacheMySQLPHPTestResults.pdf) revealed that ****Apache is bandwidth limited****, ****PHP is CPU limited****, and ****MySQL is memory limited**** and disk I/O bound.

### Analyze your site's performance bottlenecks: CPU, memory, bandwidth, input/output. Once you have identified what is causing your system performance problems you can make configuration changes or resource upgrades. Use top and ps for analysing processes that are using up too much RAM or CPU. Netstat, [mrtg](http://oss.oetiker.ch/mrtg/)/[awstats](http://awstats.sourceforge.net/), who's online block will help for identifying network problems.

### Bad guests consuming too many resources: Crawlers, Aggregators, Spammers. Search engines may crawl your site and cause a performance degradation, although if they crawl as an anonymous user then they should received cached pages which consume less resources. Slow down the robots crawling your site by adding a robots.txt

### [Apache performance](http://httpd.apache.org/docs/2.4/misc/perf-tuning.html) Apache is usually bandwidth limited so you should be aware how much bandwidth your server has access to. You can ask your hosting provider how much bandwidth you have or you can use a tool like [lperf](http://sourceforge.net/projects/iperf). The mod\_rewrite module for Apache can consume resources if the directives are included in local directories through .htaccess files. If you have control of the Apache configuration then these directives should be moved to httpd.conf. You can also configure Apache to handle more connections with [MaxSpareServers](http://httpd.apache.org/docs/2.0/mod/prefork.html" \l "maxspareservers), [ServerLimit](http://httpd.apache.org/docs/2.0/mod/mpm_common.html" \l "serverlimit), [MaxClients](http://httpd.apache.org/docs/2.0/mod/mpm_common.html" \l "maxclients). CGI is slow. mod\_php is popular and fast, FastCGI is fast and secure. [PHP-FPM](http://php-fpm.org/about/) (PHP FastCGI Process Manager) has become increasingly popular and offers more features and greater stability than FastCGI. Read this tutorial on configuring Apache for [Configuring Apache for maximum performance.](http://linuxgazette.net/123/vishnu.html)

### [PHP tuning](https://www.drupal.org/node/2602): CPU consumption and optimizers: Use a PHP optimizer such the upcoming [Alternate PHP Cache](http://pecl.php.net/package/APC), [Zend Optimizer](http://www.zend.com/store/products/zend-optimizer.php), eAccelarator a development of mmcache PHP.

* MySQL performance tuning
* Bandwidth: Images or media are likely to be the greatest source of bandwidth consumption which will limit Apache's performance. In order to ensure images are delivered quickly the theme should explictly call the images directly to reduce the PHP overhead of looking for image files. Details will be provided in the future.
* Drupal resource consumption: [Profiling memory usage](https://www.drupal.org/node/659980), measure DB query times, and duplicate queries for modules or pages with [devel.module](https://www.drupal.org/project/devel).   
  The number of modules enabled on your site may affect performance but should be measured using the techniques outlined above. If the module has slow query times look at tuning the tables with schema changes and using the the MySQL optimization tools, such as ANALYSE and EXPLAIN. Other areas for analysis are theme resource consumption if your theme has a lot of PHP calls.
* Configuring Drupal for performance: **Cron jobs such as aggregating lots of feeds can be resource intensive**. **If the cron jobs are set too frequently there can be a consistent drain on server performance**. **Drupal has a cache that is very effective for pages served to anonymous users**. It can be found at Administer >> SIte Configuration >> Performance (admin/settings/performance)
* Server architectures for scalability and performance: separate web server and database server, MySQL replication topologies, clustering servers

# Tuning php.ini for Drupal

* [**realpath\_cache\_size**](http://us3.php.net/manual/en/ini.core.php#ini.realpath-cache-size) - Determines the size of the realpath cache to be used by PHP. This value should be increased on systems where PHP opens many files (such as Drupal), to reflect the quantity of the file operations performed. The default value of this declaration is 16k, and should be changed to a value that represents the actual amount of memory used for realpath cache.

In order to find the actual amount of memory that is used by realpath cache, you’ll need to increase the default amount that is being used currently to a larger number than 16k (try 256k) and then use a script to print the actual amount to the screen. I’ve found it helpful to create a PHP block that prints out the actual value and set it to only be visible to my particular role. Then I can check memory intensive areas as well as regular pages. Once I’m done, I’ll disable the block as it’s not needed anymore.

Set realpath cache to 256k in php.ini:

realpath\_cache\_size = 256k

* realpath\_cache\_ttl - Duration of time (in seconds) for which to cache realpath information for a given file or directory. For systems with rarely changing files (such as Drupal), consider increasing the value. Raise this to 300 or higher.

The following are settings used in a Drupal site for a member’s organization that serves approximately 1500 users and more importantly the site has roughly 30 contributed modules + CiviCRM on the backend which results in quite a number of files requested.

realpath\_cache\_size = 64k

realpath\_cache\_ttl = 3600