

A PRACTICAL GUIDE TO DATA
CACHING WITH ZEND SERVER

When asked about the most significant methods to speed up PHP applications, most experts would say “cache, cache, and cache.” There are a lot of optimizations that could be done to PHP code that would make it run faster, but the most effective method to speed up PHP execution is to simply execute less, and one of the best ways to achieve that is through caching.

INNOVATION IS ACCELERATING

Zend Server provides several caching-related features, including OPcache for byte-code caching, full-page caching using Zend Page Cache, and Zend Data Cache. They are defined as:

- **Zend OPcache** performs byte-code optimization and caching. This speeds up PHP applications by eliminating the process of reading scripts from disk and compiling them, often resulting in a performance gain of 2x – 3x times faster, depending on the application. Zend OPcache runs automatically, and installing your application on top of Zend Server is all you need to do in order to enjoy its benefits. OPcache is a Zend contribution to open source.
- **Zend Page Cache** allows caching of entire PHP web pages, or more correctly put, entire HTTP responses sent out from PHP. Page Cache dramatically improves the performance of pages on web applications (sometimes running as high as 10x, 20x, and even 100x times faster), while maintaining dynamic capabilities through an elaborate system of caching rules that could be based on request parameters and user session data. Page Cache also has the benefit of not requiring any code changes, and can be set up from the Zend Server UI.
- **Zend Data Cache** is a set of API functions enabling a developer to store and manage data items (PHP strings, arrays, and other data) and even output elements in either disk-based cache or shared memory cache. Zend Data Cache allows for precision-guided caching when page caching is not an option. The provided API is easy-to-use on existing code, and in many cases a developer can skip existing code sections by simply wrapping them with caching APIs.

This white paper focuses on the Zend Data Cache API feature and demonstrates how to use it in order to speed up a typical PHP application.

HOW TO GET STARTED

Getting started with Zend Data Cache is extremely easy. The first functions you need to know about are `zend_shm_cache_fetch()` and `zend_shm_cache_store()`. These two functions are used for storing data in the shared-memory cache, and for fetching data from the cache.

Note: Zend Data Cache offers two storage modules: the shared memory (“shm”) module and the disk module. We will revisit the differences later on, for now, all functions mentioned here with the ‘zend_shm’ prefix are used for shared memory storage. For each function mentioned, there is an exact twin with the ‘zend_disk’ prefix that performs the same actions for the disk-based cache. For the two functions mentioned above, there are the `zend_disk_cache_store()` and `zend_disk_cache_fetch()` twin functions. For convenience, we will only mention the shared memory functions here however, keep in mind that you can (and should, in some situations) use the disk-based cache functions as well.

Typical code example

Code speaks louder than words, and to illustrate data caching consider the following piece of code taken from an imaginary but typical blogging platform.

```
1. function getRecentPosts($count = 5) {
2.     // Get the PDO adapter for the DB
3.     $dbAdapter = myBlogDb::getPdoAdapter();
4.
5.     // Build the query
6.     $query = "SELECT title, author, pubdate FROM posts" .
7.             " ORDER BY pubdate DESC LIMIT ?";
8.     $dbStatement = $dbAdapter->prepare($query);
9.
10.    // Execute the query and put data into an array
11.    $recentPosts = array();
12.    if ($dbStatement->execute(array($count))) {
13.        while($post = $dbStatement->fetch(PDO::FETCH_ASSOC)) {
14.            $recentPosts[] = $post;
15.        }
16.    }
17.
18.    return $recentPosts;
19. }
```

This function is used to fetch an array of the most recent posts on the blog, and will fetch the title, author, and publication date for each one. This is done by running an SQL query and placing the results into an array.

While the list of recent posts in a blog is dynamic, it is rarely changing (“rarely” is of course a relative term — but even in a very busy blog, you are likely to get hundreds or thousands of “reads” for each new post). This makes it a very good candidate for caching, it and should be fairly easy to do.

```
1. function getRecentPosts($count = 5) {
2.     // Try to fetch from the cache first
3.     $recentPosts = zend_shm_cache_fetch('recentposts');
4.
5.     if ($recentPosts === false) {
6.         // Get the PDO adapter for the DB
7.         $dbAdapter = myBlogDb::getPdoAdapter();
8.
9.         // Build the query
10.        $query = "SELECT title, author, pubdate FROM posts" .
11.                " ORDER BY pubdate DESC LIMIT ?";
12.        $dbStatement = $dbAdapter->prepare($query);
```

```

13.
14.     // Execute the query
15.     $recentPosts = array();
16.     if ($dbStatement->execute(array($count))) {
17.         while($post = $dbStatement->fetch(PDO::FETCH_ASOC)) {
18.             $recentPosts[] = $post;
19.         }
20.
21.         // Store the results in cache
22.         zend_shm_cache_store('recentposts', $recentPosts,
23.             24 * 3600);
24.     }
25. }
26.
27. return $recentPosts;
28. }

```

What has changed?

After adding three lines of code (not including blank lines and comments), we see the following happen:

- First, we try to fetch the list of recent posts from the shared memory cache. The list of recent posts is stored under a unique key — 'recentposts', which is passed as a parameter to the `fetch()` function (line 3).
- If the item does not exist, or is "stale" (in caching terms that means it has passed its expiry date), the function will return false. This allows us to check if valid data was returned from the cache (line 5). If the return value is false, we fetch the live data from the DB, as we did before.
- After fetching live data from the DB, we make sure to store it in the cache, under the 'recentposts' key, for 24 hours (line 22). We only do this if the query was successfully executed.
- Whether the data came from the DB or from the cache, we return it as we always did. The function's signature did not change, and we do not need to modify other parts of the code.

This provides the ability to skip most of the code responsible for fetching the list of recent posts from the DB, and only execute it once a day or so. This might seem like a small victory, but remember that database queries are one of the biggest hurdles in the road for scalability and performance. Each reduced query is a win.

But wait...

You may have noticed that the function takes an argument, and might return different results depending on that argument. Ignoring this when caching the results might result in unexpected data being returned.

This can be easily solved by using the `$count` parameter as part of the cache item key:

```

29. $recentPosts = zend_shm_cache_fetch("recentposts-{$count}");

```

Of course, you will need to make the same change in line 24 when storing the data. Now, we will end up with different cached items for different values of the `$count` parameter. If we use the same function to show the last five posts in the home page, but have a special page listing the last 20 posts, they will both show the correct data.

Setting up data caching was pretty simple. But the real trick when caching is not using the API — it's actually deciding what to cache, using what storage method, and for how long.

WHAT TO CACHE

Deciding what to cache is probably one of the places where inexperienced developers go wrong. After all, “premature optimization is the root of all evil”, and it's a pretty common evil too. Many developers tend to spend hours optimizing pieces of code which they think might be problematic, but in reality account for 0.02 percent of the application's execution time.

The smartest approach to caching would be to look for the slowest spots in your application, and the best way to do that would be to use a combination of performance monitoring and profiling tools — such as Zend Server Monitor combined with Zend Studio Profiler, or by using Z-Ray. You can use these tools to detect the real bottlenecks in your application, and then proceed to eradicate them by applying one of the possible caching techniques. In order to quantify the effect of any optimization you make, go to the caching section in Zend Server admin console (available in version 2018). Check the Cache Pulse page and get immediate and accurate feedback on how effective the caching usage is. See the cache efficiency with [Pulse section](#).

Another point to keep in mind is that a ten percent speedup on a page which accounts for 80 percent of your traffic (such as the home page on some applications) is usually way more effective than a 70 percent speedup on a page which accounts for two percent of your traffic. You should almost always start with the most popular parts of your application.

HOW TO CACHE

Zend Server offers two Data Cache storage modules:

- **Zend Data Cache Shared Memory Storage** should be the preferred method of caching small data items or pieces of output. It stores the data in a block of RAM on your server, which is shared by all the different PHP processes (be it Apache processes or FastCGI processes) and hence its name. This makes it very fast, but somewhat size limited; however, unless you're storing huge amounts of data in cache, you should have no problems.
- **Zend Data Cache Disk Storage** should be used when you need a relatively fast local cache, but have relatively large amounts of data to store in it (for example long strings containing lots of HTML or XML output) and do not want to waste expensive RAM on it. The disk storage module also allows for some more advanced and elaborate hacks such as sharing the cache space between a CLI script that writes to the cache and a web application that reads from it — but those are tricky, and will not be discussed here. In some cases with Linux, this option proves itself as fast as the shared memory one.

HOW LONG TO CACHE FOR

Setting the lifetime of each item in the cache mostly depends on your application's needs. The rule of thumb is that you want to keep an item in cache as much as possible, but not more than that. Consider how often the data is likely to change, and if it changes, how important it is for you to show real-time data. As an example, an application displaying stock market quotes should probably be as real-time as possible, to the extent of not caching anything. On the other hand, caching the weather forecast for 30 minutes is probably reasonable.

Another thing to consider is the efficiency of the caching system you are using, and the size of the data. Caching large data items for very short periods of time might become inefficient. Almost all caching methods need to serialize the data before writing it to cache, and un-serialize it when reading. This process is somewhat expensive, and it becomes more expensive with large, complex data structures such as arrays or objects.

PROGRAMMATIC CACHE INVALIDATION

By now you might be thinking "Ok, this is all fine — but sometimes, I can't always know in advance when the data is going to change..!" Well, you are obviously right. Lifetime-based cache expiry is good for "read-only" sort of websites, where the data is periodically updated by the site's maintainer or by some predictable system. But all the buzz these days is about the "read/write" web, where users can contribute content whenever they want. How do you go on caching something like that? How would you cache, let's say, the comments posted on an article in the imaginary blog?

One approach is to cache for short periods of time — say 10 seconds. But this has a couple of downsides: On one hand, you still need to hit the database frequently, even when there is no real reason to. On the other hand, a user might post a comment and if it does not show up immediately (but only after the user refreshed the page eight times in five seconds) they'll probably assume your site is broken, and post again, or just leave.

There's a better way: Zend Data Cache allows you to use the API in order to invalidate the cache when required. Using Data Cache in a Zend Server cluster? No worries, this is handled for you by default. Looking back at the imaginary blog example, and take a look at the function which is used to fetch a particular post's comments:

```
1. function fetchComments($post) {
2.     // Try to fetch from the cache first
3.     $comments = zend_shm_cache_fetch("comments-{$post}");
4.
5.     if ($comments === false) {
6.         // Get the PDO adapter for the DB
7.         $dbAdapter = myBlogDb::getPdoAdapter();
8.
9.         // Build the query
10.        $query = "SELECT title, author, pubdate" .
11.            " FROM comments WHERE post = ? ORDER BY pubdate";
12.        $dbStatement = $dbAdapter->prepare($query);
13.
14.        // Execute the query
15.        $comments = array();
```

```

16.     if ($dbStatement->execute(array($post))) {
17.         while($comment = $dbStatement->fetch(PDO::FETCH_ASSOC)){
18.             $comments[] = $comment;
19.         }
20.
21.         // Store the results in cache
22.         zend_shm_cache_store("comments-{$post}", $comments,
23.             24 * 3600);
24.     }
25. }
26.
27.     return $comments;
28. }

```

As you can see, the code already utilizes the caching functions, and caches the comments of each post for 24 hours. But what if someone posts a new comment? Will they need to wait for tomorrow before being able to see their new comment?

This is where cache invalidation comes into play. This technique is allowed through the `zend_shm_cache_delete()` (or `zend_disk_cache_delete()` functions). All we need to do is make sure to invalidate the cached comments for a post when a new comment is posted. Then, on the next request to the page, a "live" list of comments will be presented and stored in the cache, including the newly posted comment. The following code shows what the comment posting function will look like with this method applied:

```

1.  function postComments($post, $author, $email, $text)
2.  {
3.      // Get the PDO adapter for the DB
4.      $dbAdapter = myBlogDb::getPdoAdapter();
5.
6.      // Build the query
7.      $query = "INSERT INTO " .
8.          " comments(post, author, mail, text, date)" .
9.          " VALUES(:post, :author, :mail, :text, :date)";
10.
11.     $dbStatement = $dbAdapter->prepare($query);
12.     $dbStatement->execute(array(
13.         'post'    => $post,
14.         'author' => $author,
15.         'mail'   => $email,
16.         'text'   => $text,
17.         'date'   => $_SERVER['REQUEST_TIME']
18.     ));
19.
20.     zend_shm_cache_delete("comments-{$post}");
21. }

```

Notice on line 20, all we had to do here is explicitly delete the cached comments for this post, and this makes sure that whenever a new comment is saved in the DB, it will also appear in the blog.

Zend Data Cache also allows you to clear out all cached items by calling `zend_shm_cache_clear()` with no parameters.

Note that if you prefer not to perform the deletion across the cluster, just set the second parameter as `false` while calling the API function:

```
zend_disk_cache_clear(string $namespace, bool $clusterDelete=true)
zend_disk_cache_delete(mixed $key, bool $clusterDelete=true)
```

CACHE NAMESPACING

As you apply more caching to your application, making sure you expire all the relevant items before new content is posted becomes a hassle. For example — in the example blog application, we have several places where the list of posts can appear — one is of course in the blog's front page, but we also have the list of recent posts in our sidebar, and the same list of posts as in the front page served as an XML feed, in both RSS and Atom formats. So that might end up to be three or four different items that we must invalidate whenever a new post is made. In some applications, you might need to invalidate dozens or even hundreds of items at the same time.

Fortunately, Zend Data Cache makes it all easier by enabling you to namespace cache items.

Cache Namespacing allows you to group different cache items together, and then easily invalidate all of them with one function call.

In order to group all post-listing related cache items together, we just need to add the namespace as a prefix to the cache keys, followed by a double-colon ("::") and the key itself, like so:

```
1. $recent = zend_shm_cache_fetch("postlist::recent-{$count}");
```

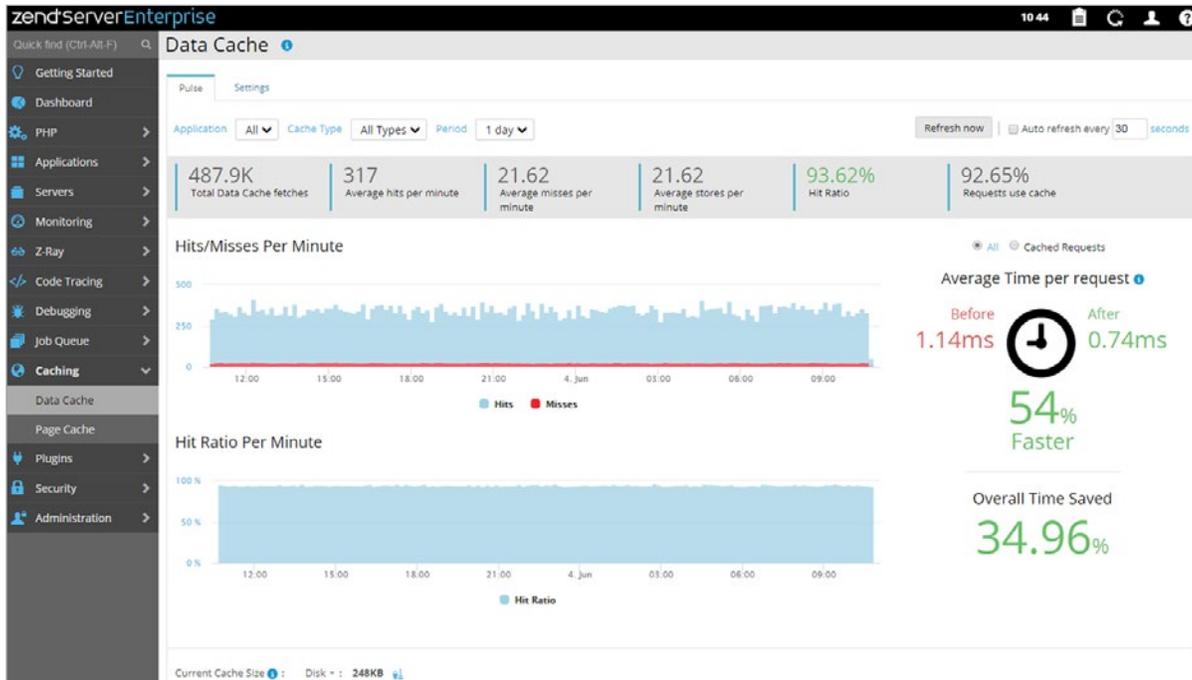
The example above modifies our `getRecentPosts()` function to use the 'postlist' namespace for when caching the recent post listing (of course, we will need to modify the `store()` calls in accordance).

With all post-listing related cache items namespaced together, we simply need to call `zend_shm_cache_clear()` with the namespace we want to clear as a parameter:

```
1. zend_shm_cache_clear("postlist");
```

CACHE EFFICIENCY WITH PULSE

The minute you start using the cache you're getting feedback in Zend Server admin console, on how much faster your application really is, what is the rate of the cache hits, and how much time was saved in server processing. Moreover, you can see how this is measured over time.



Pulse helps you fine-tune and improve the caching, based on live and historical data, such as:

- Low percentage of requests that use the cache may indicate that you aren't using caching in the right places
- Low hit ratio can indicate that you need to re-evaluate the 'time to live (TTL)' attribute, as cached pages are invalidated too quickly, which results in cache misses. You may want to consider increasing TTL and revisit Pulse later to analyze results
- Add/remove some caching in different code segments and revisit Pulse to get feedback on the change effect
- Examine the average time per request before and after using the cache. If no major improvement is evident (i.e. low % faster), it may hint that you invested time in caching where in fact it had a minimal impact

And much more.

SUMMARY

In this white paper, we covered the different functions provided by Zend Data Cache, and demonstrated how to use them in order to store, fetch, and delete items from the cache. We also covered cache namespacing, and how the API can be used to easily clear entire cache namespaces or all cached items with a single function call, across the cluster.

Additionally, we overviewed the various considerations when creating a caching strategy — what to cache, how to cache it and for how long, and the pros and cons of different cache storage methods. This information will help you to get started with Zend Data Cache.

If you are interested in learning more about the Zend Data Cache API, you can read more in the relevant chapters in the [Zend Server User Guide](#).

If you have questions, you are welcome to post them on the [Zend Server forum](#).



Rogue Wave Software is the largest independent provider of cross-platform software development tools, components, and platforms in the world. Through decades of solving the most complex problems across financial services, telecommunications, healthcare, government, academia, and other industries, Rogue Wave tools, components, platforms, and services enable developers to write better code, faster. roguewave.com