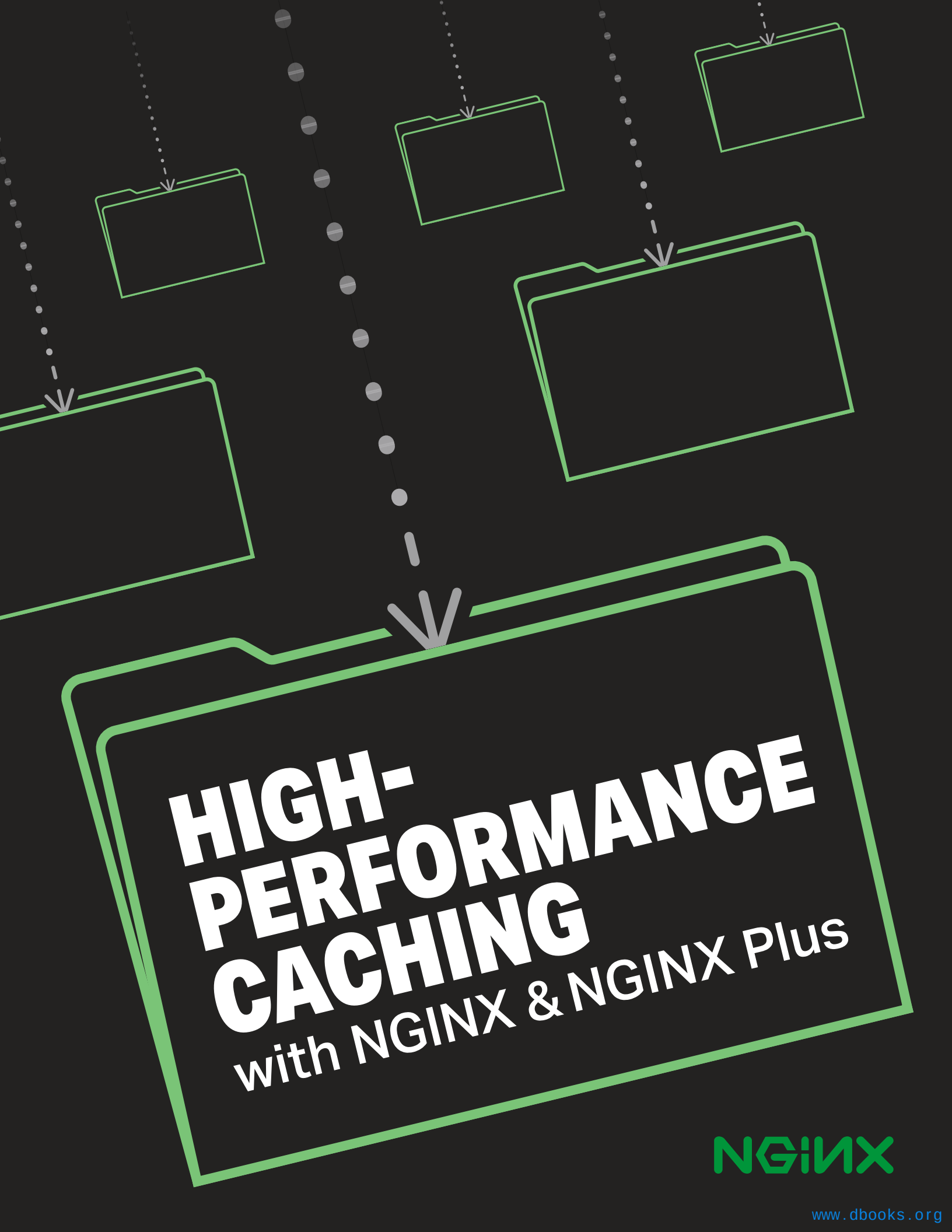
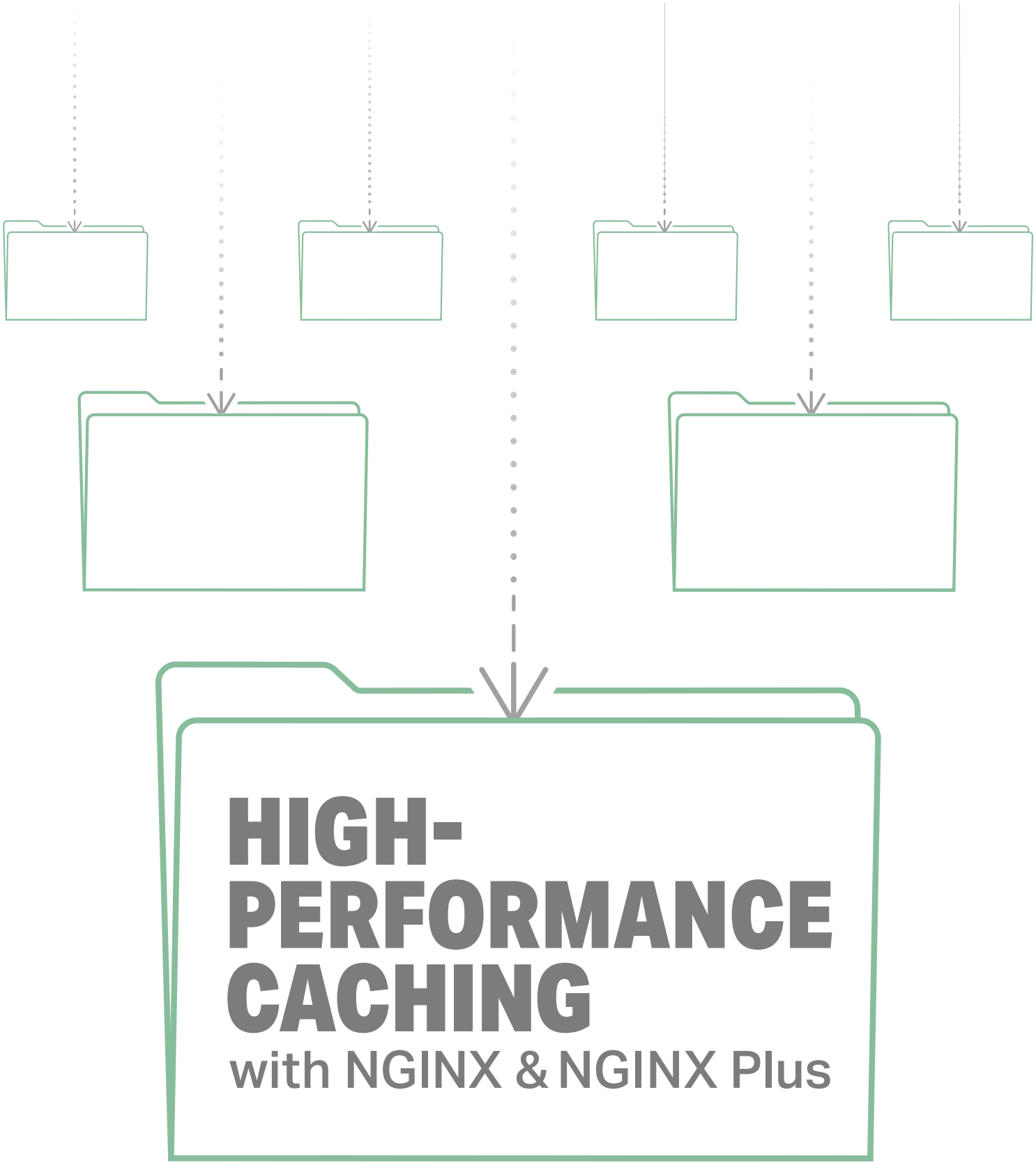


HIGH-PERFORMANCE CACHING

with NGINX & NGINX Plus

NGINX

www.dbooks.org



HIGH-PERFORMANCE CACHING

with NGINX & NGINX Plus

by Floyd Smith

NGINX

NGINX and NGINX Plus are registered trademarks of NGINX, Inc.
All other trademarks listed in this document are the property of their respective owners.

Table of Contents

www.dbooks.org

Preface

NGINX is the leading web server and reverse proxy server for high-performance websites, in use by more than 50% of the top 100,000 websites.

One of the most important capabilities of NGINX is content caching, which is a highly effective way to improve the performance of a website. In this ebook, we describe how NGINX caches, how to use caching and cache clustering, and some of the ways that you can improve performance.

We'll do a deep dive into how content caching really works so that you're equipped to debug and diagnose what's happening within NGINX. We also round up some clever hints and tips to give you fine-grained control over what NGINX does with content that can be cached, and point you to additional resources to take caching even further.

Caching takes the burden of serving and generating repetitive content away from your upstream servers, so they're freed up to run the applications that your business really needs. This gives your end users a better level of service and increases the reliability of your service as a whole in the face of big spikes of traffic from the Internet and, potentially, failures of upstream servers.

This ebook is adapted from a wide range of caching-related resources available on the NGINX website. See the Resources section at the end of this ebook for more information and links.

1 Overview

You create web applications to deliver services to users. Your application delivery infrastructure makes a huge difference in the performance and reliability that your users experience and, ultimately, in the success or failure of your business or organization.

You need to justify the investments you make to improve performance. Here you'll find important justifications for your efforts to improve web application performance through caching and other optimizations.

Why is Page Speed Important?

Web page speed is really, really important. For years, analysts have been monitoring user behavior, and have come up with what's colloquially known as the "N second rule". This is how long an average user is prepared to wait for a page to load and render before he or she gets bored and impatient and moves to a different website, to a competitor:

- 10-second rule (Jakob Nielsen, March 1997)
- 8-second rule (Zona Research, June 2001)
- 4-second rule (Jupiter Research, June 2006)
- 3-second rule (PhocusWright, March 2010)

As standards have improved and user expectations have gotten higher and higher, the period that users are prepared to wait is dropping and dropping. You could, through some slightly dubious math, extrapolate that on and conclude that users will have a negative level of patience within a few years.

www.dbooks.org

Google Changed the Rules

> *"We want you to be able to get from one page to another as quickly as you turn the page on a book."*
>
> – Urz Holzle, Google

With Google Instant Search, when you type a search term in a search box, even before you finish typing, Google is presenting candidate search results. This illustrates the huge shift in expectations on the modern Internet. Google Instant can save two to five seconds per search. As Google itself has said, "users now expect web pages to react in the same way that turning pages in a book react" – as quickly and as seamlessly and as fluidly.

The Costs of Poor Performance

If you fail to meet expected levels of performance, then there can be significant impacts on the success of your website or web application:

- Whether it's ad click-through rates: Google themselves find that their ad clickthrough rate dropped 20% when their search pages took a half a second further to load.
- Whether it's revenue: in a deliberate attempt to investigate the effect of slow web pages, Amazon deliberately increased page load in multiples of 100ms and found that the revenues from those affected customers typically dropped by 1% for each 100ms increase.

Many other analysts, websites, and investigators reported similar effects on the metrics for a website, whether that be time on page, bounce rate, etc. Recently, Google has started taking page speed into account when they calculate page rank in search results.

What appears to count is the time to first byte – the longer it takes to get the first byte of a page load, the more heavily your page rank is penalized. A website may suffer from the situation to the extent that it's not even visited often, because it appears on page two, three, or later of Google's search results.

NGINX Caching and Your Site

The caching capabilities of NGINX allow you to improve the end-user experience by reducing time to first byte, and by making web content feel snappier and more responsive.

NGINX is so capable for caching that it is widely used by major content distribution networks (CDNs) at the core of their architecture. See this panel discussion from nginx.conf 2017 for some interesting details.

Caching is used at all levels of the Internet and the web, and we encourage you to use caching extensively on your own site, even if you are also using a CDN. Caching will help you:

- Improve end-user performance
- Consolidate and simplify your web infrastructure
- Increase the availability of application servers by offloading caching work from them
- Insulate yourself from server failures

NGINX allows you to increase server capacity by taking repetitive tasks away from the upstream servers. In fact, even for content which appears to be uncacheable (the front page of a blogging site, for example), there's merit in microcaching – caching content on the NGINX proxy just for a second or so.

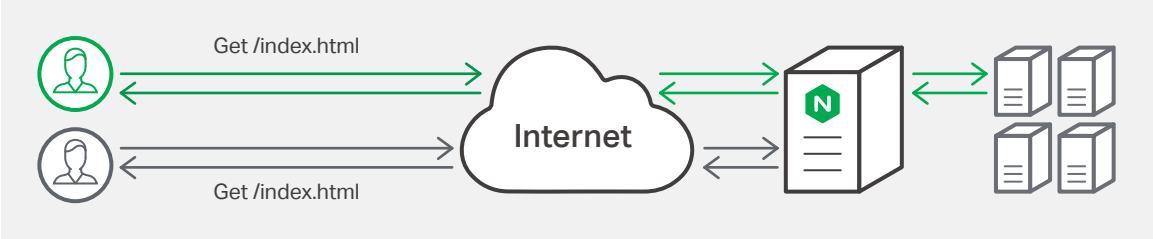
When a hundred users request the same content in the same second, NGINX will reduce that down to a single request to the origin server. NGINX will serve content back to ninety-nine of those users from its cache, with a promise that content is never more than one second out of date. That's often more than enough for a blog site or a similar website, yet makes a huge difference in performance – both in the load on the upstream servers and the expense that you have in managing and deploying sufficient capacity.

Another strategic use of NGINX is to insulate you from failures of upstream servers through the "use stale" capability. If the upstream servers are running slowly, returning errors, or experiencing some sort of fault, then NGINX can fall back to the local cached version of the content, and continue to use that until your upstream servers recover.

www.dbooks.org

2 Basic Principles of Caching

Offloading Repetitive Work



A content cache sits between a client and an "origin server" and saves copies of all the content it sees. If a client requests content that the cache has stored, it returns the content directly, without contacting the origin server. This improves performance, as the web cache is closer to the client, and makes more efficient use of the application servers, because they don't have to do the work of generating pages from scratch for many requests.

There are potentially multiple caches between the web browser and the application server: the client's browser cache, intermediary caches, content delivery networks (CDNs), and the load balancer or reverse proxy sitting in front of the application servers. Caching, including at the reverse proxy/load balancer level, can greatly improve performance.

As an example, I recently took on the task of performance-tuning a website that was loading slowly. One of the first things I noticed was that it took more than a second to generate the main home page. I discovered that, because the page was marked as not cacheable, it was being dynamically generated in response to each request.

The page itself was not changing very often and was not personalized, so this was not necessary. As an experiment, I marked the homepage to be cached for 5 seconds by the load balancer, and just doing that resulted in noticeable

improvement. The time to first byte went down to a few milliseconds, and the page loaded visibly faster.

The basic principle of content caching is to offload repetitive work from the upstream servers. When the first user requests an item of content on the website (illustrated by the green icon and green lines), his or her HTTP request is forwarded to NGINX, and from there onto the upstream server in gray on the right-hand side.

The response is forwarded back to the remote user, but if it is cacheable (and we'll talk about what that means shortly), then NGINX stores a copy of that response. When another user, the gray chap, comes along asking for the same content, then NGINX can serve that directly from its local cache rather than forging the request from the upstream server. This second scenario can then be repeated, often for many, many users.

NGINX is commonly deployed as a reverse proxy or load balancer in an application stack and has a full set of caching features. It operates as a reverse proxy cache, typically deployed in the data center or on the cloud next to the origin servers that are hosting your web content and web applications.

Static File Caching

Static file caching is a core function of NGINX. Static files usually include graphics files such as JPEGs and PNGs, and code files such as CSS and JavaScript files.

You can implement static file caching on a web server or a reverse proxy server:

- On an NGINX web server, static file caching offloads the application server; files are retrieved faster and with much less memory overhead. However, file retrieval is still being driven off the same physical server or virtual server instance, so the server's processor is still forced to deal with tasks other than running your application.
- An NGINX reverse proxy server runs on a different machine or instance from the web server, so its caching of static files consumes no resources on application servers. The application server can focus exclusively on running your application.

www.dbooks.org

There are three overall steps to implementing static file caching on NGINX:

- Specifying the root directory for searches
- Processing requests
- Optimizing response speed

On an NGINX web server, with no reverse proxy server involved, you don't cache in the usual sense. You simply redirect inquiries for static files to the web server, using the `X-Accel-Redirect` header. The application server never sees the request and can devote all its resources to application requests. With a reverse proxy server, you do use static file caching – and the physical server or virtual server instance that runs the application doesn't have any part in answering the static file requests.

As an example of optimizing response speed, the following configuration snippet enables NGINX to use the operating system's sendfile system call, which saves a step in file transmission by not copying the file to an intermediate buffer:

```
location /mp3 {
   sendfile       on;
   sendfile_max_chunk 1m;
   # ...
}
```

For specifics on configuring NGINX for static file caching, see the NGINX Plus Admin Guide.

3 How to Configure Caching

Basic Caching

To enable basic caching functionality, you only need two directives: `proxy_cache_path` and `proxy_cache`. The `proxy_cache_path` directive sets the path and configuration of the cache, and the `proxy_cache` directive activates it.

```
proxy_cache_path /path/to/cache levels=1:2
keys_zone=my_cache: 10m max_size=10g inactive=60m
use_temp_path=off;

server {
   # ...
   location / {
      proxy_cache my_cache;
      proxy_pass http://my_upstream;
   }
}
```

The parameters to the `proxy_cache_path` directive define the following settings:

- The local disk directory for the cache is called **/path/to/cache/**.
- `levels` sets up a two-level directory hierarchy under **/path/to/cache/**. Having a large number of files in a single directory can slow down file access, so we recommend a two-level directory hierarchy for most deployments. If the `levels` parameter is not included, NGINX puts all files in the same directory.
- `keys_zone` sets up a shared memory zone for storing the cache keys and metadata such as usage timers. Having a copy of the keys in memory enables NGINX to quickly determine if a request is a HIT or a MISS without having to go to disk, greatly speeding up the check. A 1 MB zone can store data for about 8,000 keys, so the 10 MB zone configured in the example can store data for about 80,000 keys.

www.dbooks.org

- `max_size` sets the upper limit of the size of the cache (to 10 GB, in this example). It's optional; not specifying a value allows the cache to grow to use all available disk space. When the cache size reaches the limit, a process called the cache manager removes the files that were least recently used to bring the cache size back under the limit.
- `inactive` specifies how long an item can remain in the cache without being accessed. In this example, a file that has not been requested for 60 minutes is automatically deleted from the cache by the cache manager process, regardless of whether or not it has expired. The default value is 10 minutes (10m). Inactive content differs from expired content. NGINX does not automatically delete content that has expired as defined by a cache control header (`Cache-Control:max-age=120` for example). Expired (stale) content is deleted only when it has not been accessed for the time specified by `inactive`. When expired content is accessed, NGINX refreshes it from the origin server and resets the `inactive` timer.
- NGINX first writes files that are destined for the cache to a temporary storage area, and the `use_temp_path=off` directive instructs NGINX to write them to the same directories where they will be cached. We recommend that you set this parameter to off to avoid unnecessary copying of data between file systems.

And finally, the `proxy_cache` directive activates caching of all content that matches the URL of the parent location block (in the example, `/`). You can also include the `proxy_cache` directive in a server block; it applies to all location blocks for the server that don't have their own `proxy_cache` directive.

Delivering Cached Content When the Origin is Down

A powerful feature of NGINX content caching is that NGINX can be configured to deliver stale content from its cache when it can't get updated content from the origin servers. This can happen if all the origin servers for a cached resource are down or temporarily busy. Rather than relay the error to the client, NGINX delivers the stale version of the file from its cache. This provides an extra level of fault tolerance for the servers that NGINX is proxying and ensures uptime in the case of server failures or traffic spikes.

To enable this functionality, include the `proxy_cache_use_stale` directive:

```
location / {
   # ...
   proxy_cache_use_stale error timeout http_500
   http_502 http_503 http_504;
}
```

With this sample configuration, if NGINX receives an error, timeout, or any of the specified 5xx errors from the origin server and it has a stale version of the requested file in its cache, it delivers the stale file instead of relaying the error to the client.

Note: With NGINX 1.11.10 and NGINX Plus R12, you can also use the stale-if-error extension of the `Cache-Control` header field to permit using a stale cached response in case of an error.

Fine-Tuning the Cache

NGINX has a wealth of optional settings for fine-tuning the performance of the cache. Here is an example that activates a few of them:

```
proxy_cache_path /path/to/cache levels=1:2 keys_zone=my_
cache:10m max_size=10g
          inactive=60m use_temp_path=off;

server {
   # ...
   location / {
      proxy_cache my_cache;
      proxy_cache_revalidate on;
      proxy_cache_min_uses 3;
      proxy_cache_use_stale error timeout updating
           http_500 http_502 http_503 http_504;
      proxy_cache_lock on;
      proxy_pass http://my_upstream;
   }
```


www.dbooks.org

These directives configure the following behavior:

- `proxy_cache_revalidate` instructs NGINX to use conditional GET requests when refreshing content from the origin servers. If a client requests an item that is cached but expired, as defined by the cache control headers, NGINX includes the `If-Modified-Since` field in the header of the GET request it sends to the origin server. This saves on bandwidth, because the server sends the full item only if it has been modified since the time recorded in the `Last-Modified` header attached to the file when NGINX originally cached it.
- `proxy_cache_min_uses` sets the number of times an item must be requested by clients before NGINX caches it. This is useful if the cache is constantly filling up, as it ensures that only the most frequently accessed items are added to the cache. By default, `proxy_cache_min_uses` is set to 1.
- The `updating` parameter to the `proxy_cache_use_stale` directive instructs NGINX to deliver stale content when clients request an item while an update to it is being downloaded from the origin server, instead of forwarding repeated requests to the server. The stale file is returned for all requests until the updated file is fully downloaded.
- With `proxy_cache_lock` enabled, if multiple clients request a file that is not current in the cache (a MISS), only the first of those requests is allowed through to the origin server. The remaining requests wait for that request to be satisfied and then pull the file from the cache. Without `proxy_cache_lock` enabled, all requests that result in cache misses go straight to the origin server.

Note: With NGINX 1.11.10 and NGINX Plus R12 and later, you can also use the stale-while-revalidate extension of the `Cache-Control` header field to permit using a stale cached response while content is being updated.

Splitting the Cache Across Multiple Hard Drives

With NGINX, there's no need to build a redundant array of inexpensive disks (RAID). If there are multiple hard drives, NGINX can be used to split the cache across them. Here is an example that splits clients evenly across two hard drives based on the request URI:

```
proxy_cache_path /path/to/hdd1 levels=1:2 keys_
zone=my_cache_hdd1:10m
                max_size=10g inactive=60m use_temp_
path=off;

proxy_cache_path /path/to/hdd2 levels=1:2 keys_
zone=my_cache_hdd2:10m
          max_size=10g inactive=60m use_temp_path=off;

split_clients $request_uri $my_cache {
   50%     "my_cache_hdd1";
   50% levels my_cache_hdd2";
}

server {
   # ...
      location / {
           proxy_cache $my_cache;
           proxy_pass http://my_upstream;
   }
}
```

The two `proxy_cache_path` directives define two caches (`my_cache_hdd1` and `my_cache_hdd2`) on two different hard drives. The `split_clients` configuration block specifies that the results from half the requests (50%) are cached in `my_cache_hdd1` and the other half in `my_cache_hdd2`. The hash based on the `$request_uri` variable (the request URI) determines which cache is used for each request, the result being that requests for a given URI are always cached in the same cache.

www.dbooks.org

4 Controlling Caching

So we've looked at how caching works, we've looked at the implementation within NGINX, and done a deep dive as to how caching stores files on disk. Now let's get a little bit smarter about caching.

For simple sites, you can turn caching on and generally it will do precisely what it needs to do to keep giving you the level of performance and the cache behavior that you want. But there are always optimizations to be made, and there are often situations where the default behavior doesn't match the behavior that you want.

Perhaps your origin servers are not setting the correct response headers, or perhaps you want to override what they're specifying inside NGINX itself. There are a myriad of ways you can configure NGINX to fine-tune how caching operates.

Delayed Caching

You can delay caching using two directives that relate to cache revalidation:

- `proxy_cache_min_uses number;`
  (saves on disk writes for very cool caches)
- `proxy_cache_revalidate on;`
  (saves on upstream bandwidth and disk writes)

Delaying caching is a very common need if you have a large corpus of content, much of which is only accessed once or twice in an hour or a day. In that case, if you have your company brochure that very few people read, it's often a waste of time to try to cache that content.

Delayed caching allows you to put a watermark in place. It will only store a cached version of this content if it's been requested a certain number of times. Until you've hit that `proxy_cache_min_uses` watermark, it won't store a version in the cache.

This allows you to exercise more discrimination for what content goes in your cache. The cache itself is a limited resource, typically bounded by the amount of memory that you have in your server, because you'll want to ensure that the cache is paged into memory as much as possible. So you often want to limit certain types of content and only put the popular requests in your cache.

Cache revalidation modifies the `If-Modified-Since` capability so that, when NGINX needs to refresh a value which has been cached, it doesn't make a simple GET to get a new version of that content; it makes a conditional GET, saying, "I have a cached version that was modified on this particular time and date".

The origin server has the option of responding with `304 Not Modified`, effectively saying the version you have is still the most recent version there is. That saves on upstream bandwidth; the origin server doesn't have to re-send content that hasn't changed, and it saves potentially on disk writes as well. NGINX doesn't have to stream that content to the disk and then swap it into place, overwriting the old version.

Control Over Cache Time

You have fine-grained control over how long content should be cached for. Quite often, the origin server will serve content up with cache headers that are appropriate for the browsers – long-term caching with a relatively frequent request to refresh the content. However, you might like the NGINX proxy sitting directly in front of your origin server to refresh files a little bit more often, to pick up changes more quickly.

There's a huge increase in load if you reduce the cache timeout for the browsers from 60 seconds to 10 seconds, but there's a very small increase in load if you reduce the cache timeout in NGINX from 60 seconds to 10 seconds. For each request, that will add five more requests per minute to your origin servers. By contrast, with the remote clients, it all depends on the number of clients with similar requests active on your site.

So, you can override the logic and the intent that your origin server specifies. You can mask out or tell NGINX to ignore certain headers: `X-Accel-Expires`, `Cache-Control`, or `Expires`. And you can provide a default cache time using the `proxy_cache_valid` directive in your NGINX configuration.

www.dbooks.org

Examples include:

- `proxy_cache_valid 200 302 10m;`
- `proxy_cache_valid 404 1m;`

Priority for headers is:

- `X-Accel-Expires`
- `Cache-Control`
- `Expires`
- `proxy_cache_valid`

The `Set-Cookie` response header means no caching.

Cache / Don't Cache

Sometimes you may not cache content that the origin server says is cacheable, or you may want to ensure that you bypass the version of content stored in NGINX. The `proxy_cache_bypass` and `proxy_no_cache` directives give you that degree of control.

You can use them as a shortcut to say that if any of a certain set of request headers are set, such as HTTP authorization, or a request parameter is present, then you want to bypass the cache – either to automatically update the cache in NGINX, or to just skip the cache completely and always retrieve content from the origin server.

Typically these are done for fairly complex caching decisions, where you're making fine-grained decisions over the values of cookies and authorization headers to control what should be cached, what should always be received from the origin server, and what should never be stored in the NGINX cache.

For example:

- `proxy_cache_bypass string ...;`
  (go to origin; may cache result)
- `proxy_no_cache string ...;`
  (if we go to origin, don't cache result)
- `proxy_no_cache $cookie_nocache $arg_nocache $http_authorization;`
  (typically used with a complex cache key, and only if the origin des not send appropriate cache-control responses)

Multiple Caches

Finally, for very large-scale deployments, you might want to explore multiple caches within an individual NGINX instance, for a couple of reasons. You might have different cache policies for different tenants on your NGINX proxy, depending on the nature of your site and depending on the importance of the performance of that site – even depending on the particular plan that each tenant is signed up for in a shared hosting situation.

Or you may have multiple disks in the NGINX host and it's most efficient to deploy an individual cache on each disk. The golden rule is to minimize the number of copies from one disk to another, and you can do that by pinning a cache to each disk and pinning the temp file for each proxy that uses that cache to the correct disk.

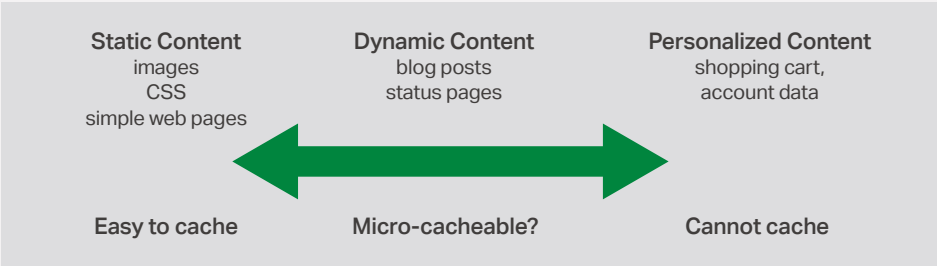
The standard operation is that when NGINX receives content from an upstream proxy, it will stream that content to disk unless it's sufficiently small and it fits in memory. Then, once that content streams to disk, it will move it into the cache. That operation is infinitely more efficient if the location on cache for the temp files (the disk where the temp files are stored) is the same as the disk where the cache is stored.

Here are examples using `proxy_cache_path` and `proxy_temp_path`:

- `proxy_cache_path /tmp/cache1 keys+zone=one:10m levels=1:2 inactive=60s;`
- `proxy_cache_path /tmp/cache1 keys+zone=one:10m levels=1:2 inactive=60s;`
- `proxy_temp_path path [level1 [level2 [level3]]];`

www.dbooks.org

5 Microcaching with NGINX



Cache content for a short time, as little as 1 second

Site content is out of date for max 1 second

Significant performance gains even for that short of a time

In microcaching, you cache content that's somewhere in the middle here on the scale – between static content (such as images, CSS, stuff that's typically easily cached and quite often put into some sort of CDN) and stuff that's all the way on the right. That's personalized content, your shopping cart, your account data, stuff that you do not ever want to cache.

Between those two extremes, there's a whole load of content (blog posts, status pages) that we consider dynamic content, but dynamic content that is cacheable. By microcaching this content, which is to say caching it for a short amount of time – as little as 1 second – we can get significant performance gains and reduce the load on the servers.

The nice part of microcaching is you can set your microcache validity to, for example, one second, and your site content will be out of date for a maximum of one second. That takes away a lot of the typical worries you have with caching, such as accidentally presenting old stale content to your users, or having to go through the hassle of validating and purging the cache every time you make a change. If you're doing microcaching, updates can be taken care of just by leveraging existing caching mechanisms.

Configuring NGINX for Microcaching

Here's a very simple NGINX configuration that enables caching.

```
proxy_cache_path /tmp/cache keys_zone=my_cache:10m
levels=1:2 inactive=600s max_size=100m;

server {
     proxy_cache cache;
     proxy_cache_valid 200 1s;
     ...

}
```

This code caches 200 responses for 1 second. `proxy _ cache _ valid` is the magic line here. It takes all the content that is cacheable and says that, for this content, we're going to cache it for one second, and then we're going to mark it as stale.

There's another directive on the top – `proxy_cache_path;` specifically, its `inactive` parameter – that says if content is inactive for 600 seconds (or in other words, no one has accessed it for 10 minutes) it's going to be deleted from the NGINX cache, whether it's stale or not.

So there are two separate concepts here. There's the idea of stale cache. This is set by the `proxy_cache_valid` directive and marks an entry as being stale but does not delete it off NGINX. It just sits there as stale content, which can be desirable. We'll see why that matters in a few minutes.

And there's the `inactive` parameter, which deletes content – whether or not it's stale – if it has not been accessed by a user for a given amount of time; in this case, 10 minutes.

www.dbooks.org

Optimized Microcaching with NGINX

That's how we enable microcaching on a basic level within NGINX. Then there are some directives that we determined, based on testing, can improve performance even more in the microcaching case. (Some of these directives were described earlier in the book, but they're described here in a microcaching context.)

```
server {
   proxy_cache one;
   proxy_cache_lock on;
   proxy_cache_valid 200 1s;
   proxy_cache_use_stale updating;
   # ...
}
```

proxy_cache_lock
The first directive, `proxy_cache_lock`, says that if there are multiple simultaneous requests for the same uncached or stale content – in other words, content that has be refreshed – only the first of those requests is allowed through, and subsequent requests for the same content are queued up. That way, when the first request is satisfied, the other guys will get it from the cache. This saves a significant amount of work on the backend.

proxy_cache_valid
The `proxy_cache_valid` directive marks the entry as being stale, but does not clear it. This makes the entry available for use in microcaching.

proxy_cache_use_stale
`proxy_cache_use_stale` allows you to serve stale content in various scenarios when fresh content is not available. In this case, we're restricting it to updating, so we're instructing NGINX to serve stale content when a cached entry is in the process of being updated.

That's microcaching with NGINX in a nutshell. If you want more details on microcaching with NGINX, see our blog.

6 The Inner Workings of Caching in NGINX

Now, having looked at how we can examine contact caching from the outside, let's have a look at it from inside. How does it function within NGINX? As I mentioned earlier, the content cache in NGINX functions in much the same way as files on disk are handled. You get the same performance, the same reliability, and the same operating system optimizations when you're serving content from your content cache as you do when you serve static content – the performance that NGINX is renowned for.

NGINX uses a persistent disk-based cache; the operating system page cache keeps content in memory, with hints from NGINX processes. In this chapter, we'll look at:

- How content is stored in the cache
- How the cache is loaded at startup
- Pruning the cache over time
- Purging content manually from the cache

The content cache is stored on disk in a persistent cache. We work in conjunction with the operating system to swap that disk cache into memory, providing hints to the operating system page cache as to what content should be stored in memory. This means that when we need to serve content from the cache, we can do so extremely quickly.

The metadata around the cache, information about what is there and its expiration time, is stored separately in a shared memory section across all the NGINX processes and is always present in memory. So NGINX can query the cache, search the cache, extremely fast; it only needs to go to the page cache when it needs to pull the response and serve it back to the end user.

We'll look at how content is stored in the cache, we'll look at how that persistent cache is loaded into empty NGINX worker processes on startup, we'll look at

www.dbooks.org

some of the maintenance NGINX does automatically on the cache, and we'll round up by looking at how you can manually prune content from the cache in particular situations.

How is Cached Content Stored?

You recall that the content cache is declared using a directive called `proxy_cache_path`. This directive specifies a number of parameters: where the cache is stored on your file system, the name of the cache, the size of the cache in memory for the metadata, and the size of the cache on disk. In this case there's a 40 MB cache on disk.

Here's a description of how key directives interact to get content into the cache:

```
proxy_cache_path /tmp/cache keys_zone=my_cache:
10m levels=1:2 max_size=40m;
```

```
proxy_cache_key $scheme$proxy_host$uri$is_args$args
```
(define the cache key)

```
$ echo –n "httplocalhost:8002/time.php" | md5sum
6d91b1ec887b7965d6a926cff19379b4
```

```
$ cat /tmp/cache/4/9b/6d91b1ec887b7965d6a926cff19379b4
```
(verify it's there)

The key to understanding where the content is stored is understanding the cache key – the unique identifier that NGINX assigns to each cacheable resource. By default that identifier is built up from the basic parameters of the request: the scheme, `Host` header, the URI, and any string arguments.

But you can extend that if you want using things like cookie values or authentication headers or even values that you've calculated at runtime. Maybe you want to store different versions for users in the UK than for users in the US. This is all made possible by configuring the `proxy_cache_key` directive.

When NGINX handles a request, it will calculate the `proxy_cache_key`, and from that value it will then calculate an MD5 sum. You can replicate that yourself using the command-line example I've shown further down the slide. We take the cache key **httplocalhost:8002/time.php** and pipe that through md5sum. (Be careful, when you're doing this from the shell, not to pipe a new line through as well.)

That will calculate the MD5 hash value that corresponds to that cacheable content. NGINX uses that hash value to calculate the location on disk that content should be stored. You'll see in the `proxy_cache_path` that we specify a two-level cache with a one-character and then a two-character directory. We pull those characters off the end of the string to create a directory called 4 and a subdirectory called 9b, and then we drop the content of the cache (plus the headers and a small amount of metadata) into a file on disk.

You can test the content caching. You can print out the cache key as one of the response headers, you can pump it through md5sum to calculate the hash correspondence of that value. Then you can inspect the value on disk, to see if it's really there, and the headers that NGINX cached, to understand how this all fits together.

Loading Cache From Disk

Now that content is stored on disk and is persistent, when NGINX starts it needs to load that content into memory – or rather, it needs to work its way through that disk cache, extract the metadata, and then load the metadata into memory in the shared memory segment used by each of the worker processes. This is done using a process called the cache loader.

A cache loader spins up at startup and runs once, loading metadata onto disk in small chunks: 100 files at a time, sandboxed to 200 ms, and then pausing for 50 ms in-between, and then repeating until it's worked its way through the entire cache and populated the shared memory segment.

The cache loader then exits and doesn't need to run again unless NGINX is restarted or reconfigured and the shared memory segment needs to be reinitialized. You can tune the operation of the cache loader, which may be appropriate if you have very fast disks and a light load. You can make it run faster; or, you might want to wind it back a little bit if you're storing a cache with a huge number of files and slow disks, and you don't want the cache loader to use excessive amounts of CPU when NGINX starts up.

www.dbooks.org

Here are a few keys to using the cache loader:

- Cache metadata stored in a shared memory segment...
- ...populated at startup from the cache by the cache loader

```
proxy_cache_path path keys_zone=name:size
[loader_files=number] [loader_threshold=time]
[loader_sleep=time]
```

Default values are 100 for the number of files, 200ms for the threshold time, and 50ms for the sleep time. If these values are used, the cache loader loads files in blocks of 100, takes no longer than 200ms, then pauses for 50ms and repeats.

Managing the Disk Cache

Once the cache is in memory, and files are stored on disk, there's a risk that cached files that are never accessed may hang around forever. NGINX will store them the first time it sees them, but if there are no more requests for a file, then the file will just sit there on disk until something comes along and cleans it out.

This something is the *cache manager*; it runs periodically, purging files from the disk that haven't been accessed within a certain period of time, and it deletes files if the cache is too big and has overflowed its declared size. It deletes them in a least-recently-used fashion.

You can configure this operation using parameters to the `proxy_cache_path` directive, just as you configure the cache loader:

- The inactive time defaults to 10 minutes.
- The `max-size` parameter has no default limit. If you impose a `max-size` limit on the cache, at times it may exceed that limit, but when the cache manager runs it will then prune the least recently used files to take it back underneath that limit.

Cache Purging with NGINX Plus

Let's talk quickly about cache purging in NGINX. There are different ways to accomplish this. You can compile a third-party module and load it into NGINX, and that's fine. But for our customers with NGINX Plus, there's a built-in cache purging module written by the same developers that made NGINX, with the backing of our support team and engineering staff.

So, for NGINX Plus it's prebundled, and this is a sample configuration that enables it. It allows you to run the `curl` command with the PURGE verb (as opposed to GET) … to delete everything with that root URL. Or, you can name a specific file to purge; for example, **www.example.com/image.jpg**. So NGINX Plus includes this powerful prebuilt cache-purging mechanism.

Finally, there are times that you may wish to purge content from disk. You want to find a file and delete it; it's relatively easy to do if you know the techniques that we talked about earlier – running the cache key through md5sum – or just running a recursive grep across the file system to identify the file or files that you need to delete.

Alternatively, if you're using NGINX Plus, you can use the cache purge capability built into that product. The cache purge capability allows you to take a particular parameter from the request; generally, we use a method called PURGE as the way to identify that it's a cache-purge request.

The following code sets up purging in NGINX Plus.

```
proxy_cache_path /tmp/cache keys_zone=my_cache:10m
levels=1:2 inactive=60s;

map $request_method $purge_method {
     PURGE 1;
     default 0;
}

server {
     proxy_cache mycache;
     proxy_cache_purge $purge_method;
}

$ curl –X PURGE –D – http://www.example.com/*

HTTP/1.1 204 No Content
```


www.dbooks.org

Purging is handled by a special NGINX Plus handler which inspects the URI and deletes all of the files that match that URI from the cache. The URI can be suffixed with an asterisk so that it becomes a stem. In this case, we're going to use the purge capability to delete every single file that's served up from localhost host port 8001, but of course you could put subdirectories as well.

Whichever method you use, at any point you are entirely safe to delete files from the cache on disk, or even `rm -rf` the entire cache directory. NGINX won't skip a beat; it'll continue to check for the presence of files on disk. If they're missing, then that creates a cache miss. NGINX will then go on and retrieve the cache from the origin server and store it back in the cache on disk. So it's always safe and reliable and stable if you need to wipe individual files from the cache.

7 Shared Caches with NGINX Cache Clusters

This chapter describes how to implement high-capacity and high-availability caching with the open source NGINX software and NGINX Plus.

Why Doesn't NGINX Use a Shared Disk for Caching?

Each NGINX server acts as an independent web cache server. There is no technical means to share a disk-based cache between multiple NGINX servers; this is a deliberate design decision.

Storing a cache on a high-latency, potentially unreliable shared filesystem is not a good design choice. NGINX is sensitive to disk latency, and even though the thread pools capability offloads `read()` and `write()` operations from the main thread, when the filesystem is slow, and cache I/O is high, then NGINX may become overwhelmed by large volumes of threads.

Maintaining a consistent, shared cache across NGINX instances would also require cluster-wide locks to synchronize overlapping cache operations such as fills, reads, and deletes. Finally, shared filesystems introduce a source of unreliability and unpredictable performance to caching, where reliability and consistent performance is paramount.

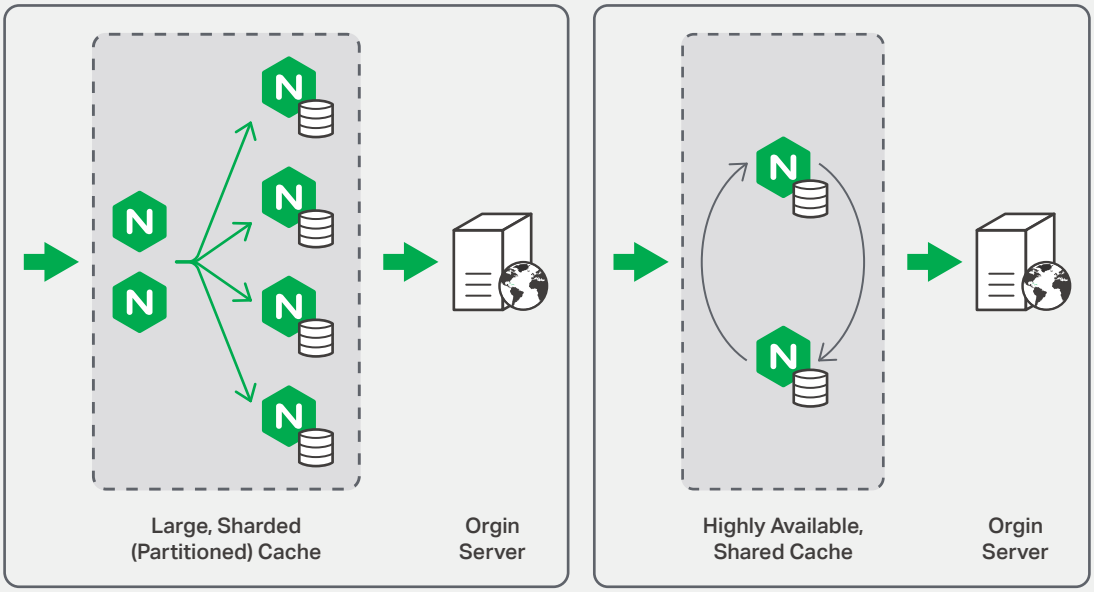
Why Share a Cache Across Multiple NGINX Servers?

Although sharing a filesystem is not a good approach for caching, there are still good reasons to cache content across multiple NGINX servers, each with a corresponding technique. This chapter will cover two techniques:

- Cache sharding – If your primary goal is to create a very high-capacity cache, shard (partition) your cache across multiple servers. We'll cover this technique in this post.

www.dbooks.org

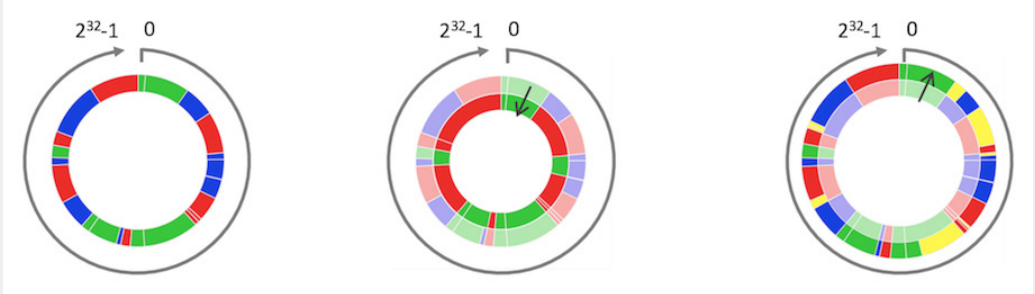
- Cache clustering – If your primary goal is to achieve high availability while minimizing load on the origin servers, use a highly available shared cache. For this technique, see the next section, Method 2: Creating a Highly Available Cache Cluster.



Method 1: Sharding Your Cache

Sharding a cache is the process of distributing cache entries across multiple web cache servers. NGINX cache sharding uses a consistent hashing algorithm to select the one cache server for each cache entry.

The figures show what happens to a cache sharded across three servers when either one server goes down or another server is added.

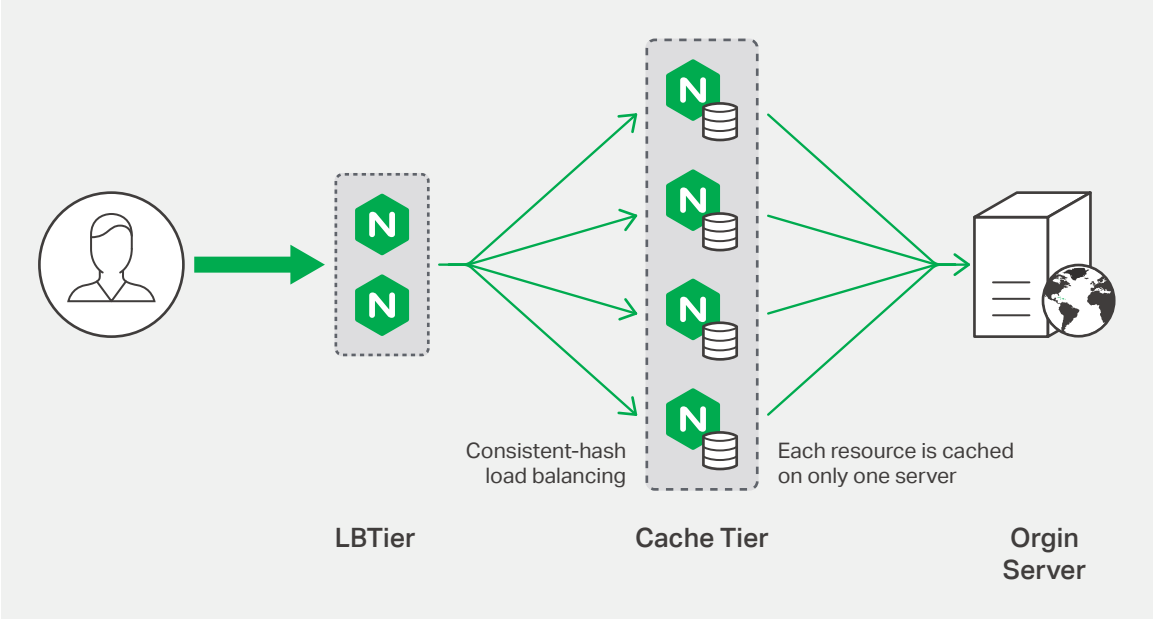


Three cache servers: each URL is assigned to the red, blue, or green server according to the hash value

If the blue cache server fails, its share is distributed between the remaining red and green servers

If a new cache server (yellow) is added, it takes share from each of the currently working servers

The total cache capacity is the sum of the cache capacity of each server. You minimize trips to the origin server because only one server attempts to cache each resource; you don't have multiple independent copies of the same resource.



This pattern is fault-tolerant in the sense that if you have ***N*** cache servers and one fails, you lose only 1/***N*** of your cache. This 'lost portion' is evenly distributed by the consistent hash across the remaining ***N***–1 servers. Simpler hashing methods instead redistribute the entire cache across the remaining servers, so you lose almost all of your cache during the redistribution.

When you perform consistent-hash load balancing, use the cache key (or a subset of the fields used to construct the key) as the key for the consistent hash:

```
upstream cache_servers {
   hash $scheme$proxy_host$request_uri consistent;
   server red.cache.example.com;
   server green.cache.example.com;
   server blue.cache.example.com;
}
```

You can distribute incoming traffic across the Load Balancer (LB) tier using the active-passive high availability solution in NGINX Plus, round-robin DNS, or a high-availability solution such as `keepalived`.

www.dbooks.org

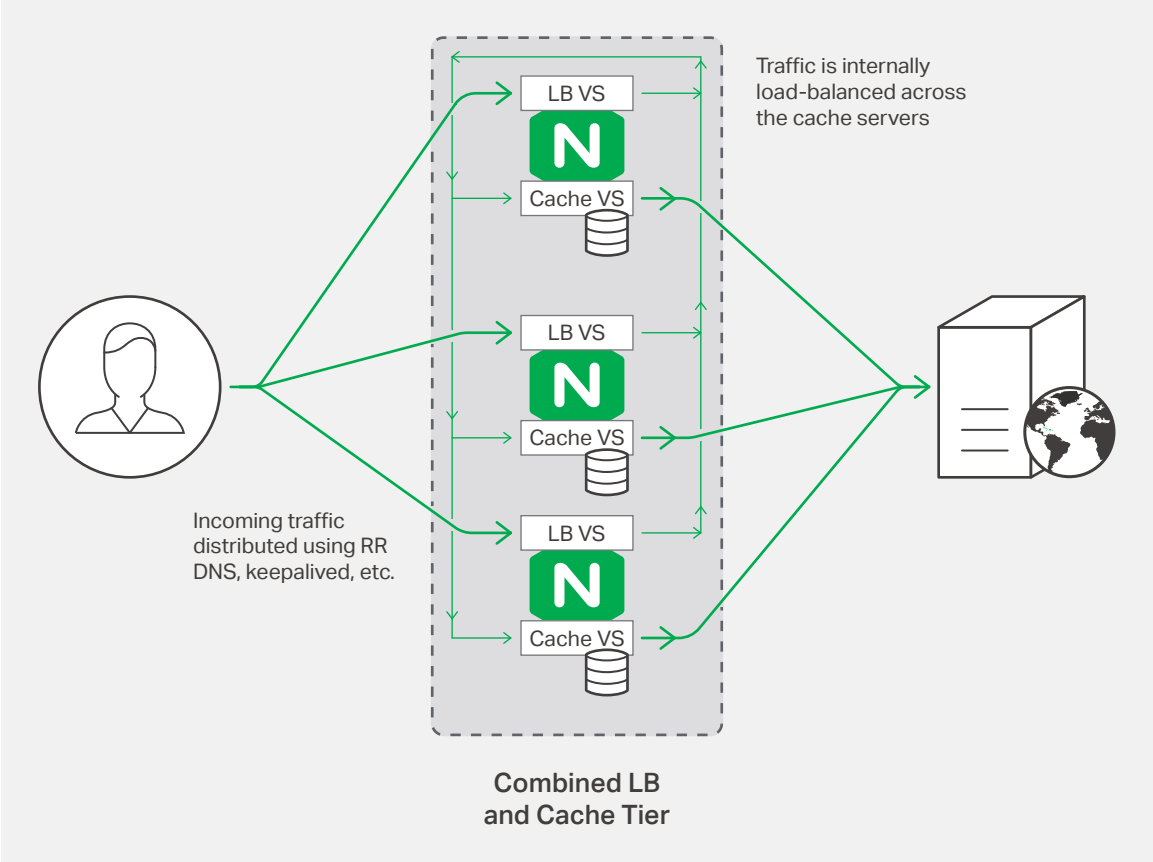
Optimizing Your Sharded Cache Configuration

You can choose either of two optimizations to your cache-sharding configuration: combining the load balancer and cache tiers, or configuring a first-level "hot" cache.

Combining the Load Balancer and Cache Tiers

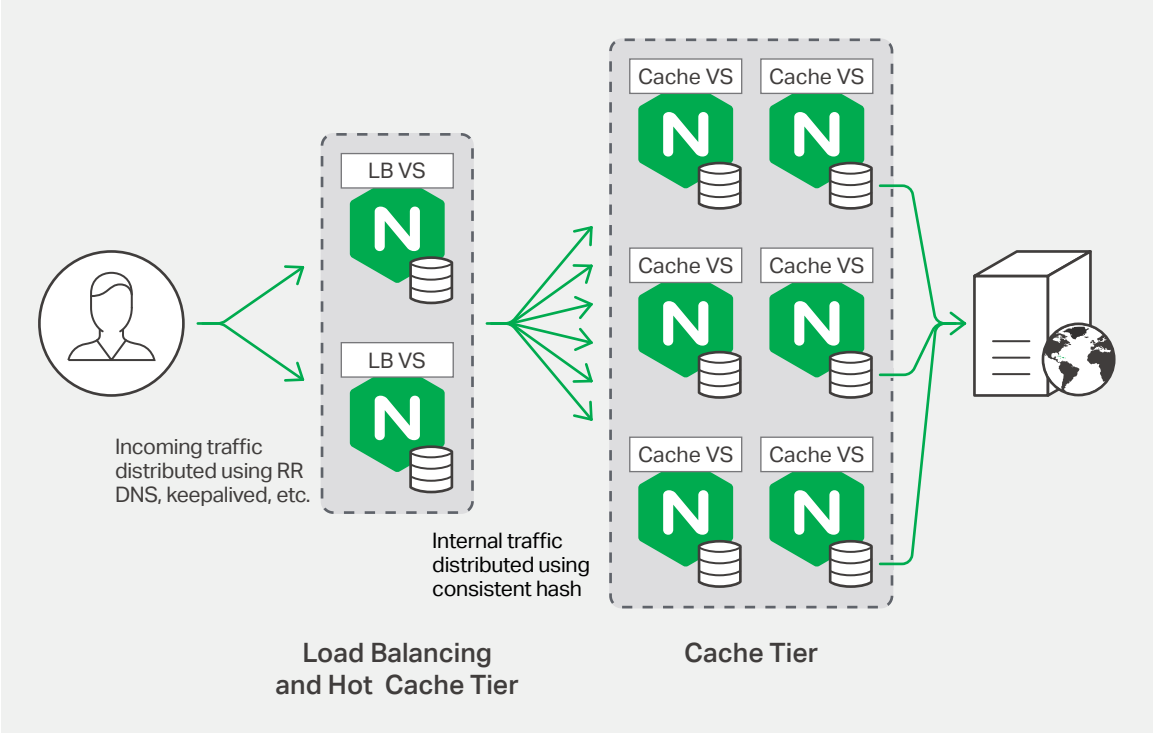
You can combine the load balancer and cache tiers. In this configuration, two virtual servers run on each NGINX instance.

The load-balancing virtual server ("LB VS" in the figure) accepts requests from external clients and uses a consistent hash to distribute them across all NGINX instances in the cluster, which are connected by an internal network. The caching virtual server ("Cache VS") on each NGINX instance listens on its internal IP address for its share of requests, forwarding them to the origin server and caching the responses. This allows all NGINX instances to act as caching servers, maximizing your cache capacity.



Configuring a First-Level "Hot" Cache

Alternatively, you can configure a first-level cache on the front-end load balancing tier for very hot content, using the large shared cache as a second-level cache. This can improve performance and reduce the impact on the origin server if a second-level cache tier fails, because content only needs to be refreshed as the first-tier cache content gradually expires.

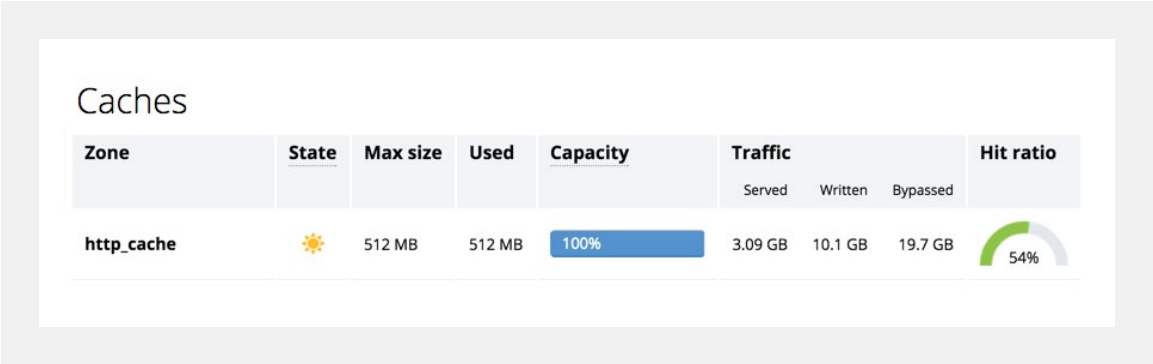


If your cache cluster is handling a very large volume of hot content, you may find that the rate of churn on the smaller, first-level cache is very high. In other words, the demand for the limited space in the cache is so high that content is evicted from the cache (to make room for more recently requested content) before it can be used to satisfy even one subsequent request.

One indicator of this situation is a low ratio of served content to written content, two metrics included in the extended statistics reported by the NGINX Plus Status module. They appear in the Served and Written fields on the Caches tab of the built-in live activity monitoring dashboard. (Note that the Status module and live activity monitoring dashboard are not available in open source NGINX.)

www.dbooks.org

This screen shot indicates the situation where NGINX is writing more content to the cache than it's serving from it:



In this case, you can fine-tune your cache to store just the most commonly requested content. The `proxy_cache_min_uses` directive can help to identify this content.

Summary of Method 1

Sharding a cache across multiple NGINX or NGINX Plus web cache servers is an effective way to create a very high-capacity, scalable cache. The consistent hash provides a good degree of high availability, ensuring that if a cache fails, only its share of the cached content is invalidated.

The next section describes an alternative shared cache pattern that replicates the cache on a pair of NGINX or NGINX Plus cache servers. Total capacity is limited to the capacity of an individual server, but the configuration is fully fault-tolerant, and no cached content is lost if a cache server becomes unavailable.

Method 2: Creating a Highly Available Cache Cluster

The previous section described a pattern for creating very large, sharded cache clusters. This section describes how to use two or more NGINX or NGINX Plus cache servers to create a highly available cache cluster.

This pattern is effective when you need to create a very large-capacity cache that can be scaled up at will. Because each resource is only cached on one server, it is not fully fault-tolerant, but the consistent-hash load balancing ensures that if a server fails, only its share of the cached content is invalidated.

If minimizing the number of requests to your origin servers at all costs is your primary goal, then the cache sharding solution is not the best option. Instead, a solution with careful configuration of primary and secondary NGINX instances can meet your requirements:

- The primary NGINX instance receives all traffic and forwards requests to the secondary instance.
- The secondary instance retrieves the content from the origin server and caches it; the primary instance also caches the response from the secondary instance and returns it to the client.

Both devices have fully populated caches, and the cache is refreshed according to your configured timeouts.

Configuring the Primary Cache Server

Configure the primary cache server to forward all requests to the secondary server and cache responses. As indicated by the `backup` parameter to the server directive in the upstream group, the primary server forwards requests directly to the origin server in the event that the secondary server fails:

```
proxy_cache_path /tmp/mycache keys_zone=mycache:10m;
server {
   status_zone mycache; # for NGINX Plus extended status

   listen 80;

   proxy_cache mycache;
   proxy_cache_valid 200 15s;

   location / {
      proxy_pass http://secondary;
   }
}

upstream secondary {
   zone secondary 128k; # for NGINX Plus extended status

   server 192.168.56.11; # secondary
   server 192.168.56.12 backup; # origin
}
```


www.dbooks.org

Configuring the Secondary Cache Server

Configure the secondary cache server to forward requests to the origin server and cache responses.

```
proxy_cache_path /tmp/mycache keys_zone=mycache:10m;

server {
   status_zone mycache; # for NGINX Plus extended status

   listen 80;

   proxy_cache mycache;
   proxy_cache_valid 200 15s;

   location / {
   proxy_pass http://origin;
   }
}

upstream origin {

   zone origin 128k; # for NGINX Plus extended status

   server 192.168.56.12; # origin
}
```

Configuring High Availability

Finally, you need to configure high availability (HA) so that the secondary server takes the incoming traffic if the primary fails; the primary takes the traffic back when it subsequently recovers.

In this example, we use the active-passive HA solution for NGINX Plus. The externally advertised virtual IP address is 192.168.56.20, and the primary acts as the master in the cluster. If you are using open source NGINX, you can manually install and configure `keepalived` or a different HA solution.

Failover Scenarios

Recall that we want to create a highly available cache cluster that continues to operate even if a cache server fails. We don't want the load on the origin server to increase, either when a cache server fails, or when it recovers and needs to refresh stale content.

Suppose the primary fails and the NGINX Plus HA solution transfers the external IP address to the secondary.

The secondary has a full cache and continues to operate as normal. There is no additional load on the origin server.

When the primary cache server recovers and starts receiving client traffic, its cache will be out of date and many entries will have expired. The primary will refresh its local cache from the secondary cache server; because the cache on the secondary server is already up-to-date, there is no increase in traffic to the origin server.

Now suppose the secondary fails. The primary detects this (using a health check configured as part of the HA solution) and forwards traffic directly to the backup server (which is the origin server).

The primary server has a full cache and continues to operate as normal. Once again, there is no additional load on the origin server.

When the secondary recovers, its cache will be out of date. However, it will only receive requests from the primary when the primary's cache expires, at which point the secondary's copy will also have expired. Even though the secondary needs to make a request for content from the origin server, this does not increase the frequency of requests to the origin. There's no adverse effect on the origin server.

Testing the Failover Behavior

To test our HA solution, we configure the origin server to log requests and to return the current time for each request. This means that the origin server's response changes every second:

```
access_log /var/log/nginx/access.log;

location / {
   return 200 "It's now $time_local\n";
}
```

The primary and secondary cache servers are already configured to cache responses with status code 200 for 15 seconds. This typically results in cache updates every 15 or 16 seconds.

```
proxy_cache_valid 200 15s;
```


www.dbooks.org

Verifying Cache Behavior

Once per second, we send an HTTP request to the highly available virtual IP address for the cache cluster. The response does not change until the caches on the primary and secondary servers expire and the response is refreshed from the origin server. This happens every 15 or 16 seconds.

```
$ while sleep 1 ; do curl http://192.168.56.20/ ; done
It's now 9/Feb/2017:06:35:03 -0800
It's now 9/Feb/2017:06:35:03 -0800
It's now 9/Feb/2017:06:35:03 -0800
It's now 9/Feb/2017:06:35:19 -0800
It's now 9/Feb/2017:06:35:19 -0800
^C
```

We can also inspect the logs on the origin server to confirm that it is receiving a request only every 15 or 16 seconds.

Verifying Failover

We can verify that the failover is working correctly by stopping either the primary or the secondary server – for example, by stopping the nginx processes. The constant-load test continues to run, and the response is consistently cached.

Inspecting the access log on the origin server confirms that it only ever receives a request every 15 to 16 seconds, no matter which of the cache servers fails or recovers.

Timing of Cache Updates

In a stable situation, the cached content is normally updated every 15 to 16 seconds. The content expires after 15 seconds, and there is a delay of up to 1 second before the next request is received, causing a cache update.

Occasionally, the cache will appear to update more slowly (up to 30 seconds between changes in content). This occurs if the primary cache server's content expires and the primary retrieves cached content that is almost expired from the secondary. If this presents a problem, you can configure a shorter cache timeout on the secondary server.

Summary of Method 2

Tiering caches between two or more NGINX cache servers in the way we've described here is an effective way to create a highly available cache cluster that minimizes the load on origin servers, protecting them from spikes of traffic when one of the cache servers fails or recovers.

The total capacity of the cache is limited to the capacity of a single cache server. Method 1, above, describes an alternative sharded cache pattern that partitions a cache across a cluster of cache servers. In that case, the total capacity is the sum of all of the cache servers, but content is not replicated to protect against spikes in traffic to the origin server if a cache server fails.

www.dbooks.org

FAQ

This section answers some frequently asked questions about NGINX content caching.

Does proxy cache revalidation factor in ETags, or just the If-Modified-Since date of the content?

The proxy cache revalidate capability is the capability that allows NGINX to make a conditional GET to the upstream server to check whether content has changed. The answer is, NGINX just checks the `If-Modified-Since` date of the content. As a point of practice, it's generally good practice to always include `If-Modified-Since` in your responses and treat ETags as optional, because they're not as consistently or as widely handled as the "last modified" date that you'll handle in response.

Is it possible for NGINX to load balance its caching for a single site between a few equal disks for best performance?

Yes it is; it takes a little bit of work. The typical scenario is to deploy a bunch of disks with no RAID, and then deploy individual caches, one pinned to each disk. It requires some additional configuration and partitioning of the traffic. If you want some help configuring that, then reach out to our community and we'll deal with your request there. If you're using NGINX Plus, reach out to our support team and we'll be delighted to help.

Are all the caching primitives and directives being respected?

In general, yes. There are a couple of edge cases, such as `Vary` headers, which are not respected. In many cases, there is a degree of latitude in how various caches interpret the requirements of the RFC. Where possible, we've gone for implementations that are reliable, consistent, and easy to configure.

Are both upstream headers and data being cached?

Yes, they are. If you receive a response from the cache, then the headers are cached as well as the response body.

Does caching work for HTTP/2?

Absolutely. You can think of it as a frontend proxy for NGINX, although in practice it's very, very deeply intertwined in the NGINX kernel. And, yes, SPDY works for caching.

Does PageSpeed use NGINX caching or its own caching mechanism?

That's a question that you would need to share with the PageSpeed developers.

How do other content caches compare to NGINX?

CDNs are very effective content caching solutions. CDNs are deployed as a service; you have more limited control over how content is cached and how it expires within that, but they are a very, very effective tool for bringing content closer to end users. NGINX is a very effective tool for accelerating the web application. Very commonly, both are deployed together. In the case of stand-alone caches such as Varnish: once again, they are very capable technologies that work in a similar fashion to NGINX in many respects. One of the benefits of NGINX is that it brings together origin-serving application gateways, caching, and load balancing into one single solution. So that gives you a simpler, more consolidated infrastructure that's easier to roll out, easier to manage, and easier to debug and diagnose if you have any issues.

Can the NGINX cache be instrumented?

Yes, with the `add_header` directive:

```
add_header X-Cache-Status $upstream_cache_status;
```

This example adds an `X-Cache-Status` HTTP header in responses to clients. The following are the possible values for `$upstream_cache_status`:

- **MISS** – The response was not found in the cache and so was fetched from an origin server. The response might then have been cached.
- **BYPASS** – The response was fetched from the origin server instead of served from the cache because the request matched a `proxy_cache_bypass` directive. The response might then have been cached.
- **EXPIRED** – The entry in the cache has expired. The response contains fresh content from the origin server.
- **STALE** – The content is stale because the origin server is not responding correctly, and `proxy_cache_use_stale` was configured.

www.dbooks.org

- **UPDATING** – The content is stale because the entry is currently being updated in response to a previous request, and `proxy_cache_use_stale` updating is configured.
- **REVALIDATED** – The `proxy_cache_revalidate` directive was enabled and NGINX verified that the current cached content was still valid (`If-Modified-Since` or `If-None-Match`).
- **HIT** – The response contains valid, fresh content direct from the cache.

How does NGINX determine whether or not to cache something?

By default, NGINX respects the `Cache-Control` headers from origin servers. It does not cache responses with `Cache-Control` set to `Private`, `No-Cache`, or `No-Store`, or with `Set-Cookie` in the response header. NGINX only caches GET and HEAD client requests. You can override these defaults, as described in the answers below.

NGINX does not cache responses if `proxy_buffering` is turned off. It is on by default.

Can Cache-Control headers be ignored?

Yes, with the `proxy_ignore_headers` directive. For example, with this configuration:

```
location /images/ {
   proxy_cache my_cache;
   proxy_ignore_headers Cache-Control;
   proxy_cache_valid any 30m;
   # ...
}
```

NGINX ignores the `Cache-Control` header for everything under **/images/**. The `proxy_cache_valid` directive enforces an expiration for the cached data, and is required if ignoring `Cache-Control` headers. NGINX does not cache files that have no expiration.

Can NGINX cache content with a Set-Cookie in the header?

Yes, with the `proxy_ignore_headers` directive, as discussed in the previous answer.

Can NGINX cache POST requests?

Yes, with the `proxy_cache_methods` directive:

```
proxy_cache_methods GET HEAD POST;
```

This example enables caching of POST requests.

Can NGINX cache dynamic content?

Yes, provided the `Cache-Control` header allows for it. Caching dynamic content for even a short period of time can reduce load on origin servers and databases, which improves time to first byte, as the page does not have to be regenerated for many requests.

Can I punch a hole through my cache?

Yes, with the `proxy_cache_bypass` directive:

```
location / {
   proxy_cache_bypass $cookie_nocache $arg_nocache;
   # ...
```

The directive defines request types for which NGINX requests content from the origin server immediately, instead of trying to find it in the cache first. This is sometimes referred to as "punching a hole" through the cache. In this example, NGINX does it for requests with a `nocache`cookie or argument, for example **http://www.example.com/?nocache=true**. NGINX can still cache the resulting response for future requests that aren't bypassed.

What cache key does NGINX use?

The default form of the keys that NGINX generates is similar to an MD5 hash of the following NGINX variables: `$scheme$proxy_host$request_uri`; the actual algorithm used is slightly more complicated.

```
proxy_cache_path /path/to/cache levels=1:2 keys_zone=my_
cache:10m max_size=10g
          inactive=60m use_temp_path=off;
server {
   # ...
   location / {
      proxy_cache my_cache;
      proxy_pass http://my_upstream;
   }
}
```


www.dbooks.org

For this sample configuration, the cache key for **http://www.example.org/my_image.jpg** is calculated as md5 ("**http://my_upstream:80/my_image.jpg**").

Note: The `$proxy_host` variable is used in the hashed value instead of the actual host name (www.example.com). `$proxy_host` is defined as the name and port of the proxied server as specified in the `proxy_pass` directive.

To change the variables (or other terms) used as the basis for the key, use the `proxy_cache_key` directive (see also the following question).

Can I use a cookie as part of my cache key?

Yes, the cache key can be configured to be any arbitrary value, for example:

```
proxy_cache_key $proxy_host$request_uri$cookie_jessionid;
```

This example incorporates the value of the **JSESSIONID** cookie into the cache key. Items with the same URI but different **JSESSIONID** values are cached separately as unique items.

Does NGINX use the ETag header?

In NGINX 1.7.3 and NGINX Plus R5 and later, the **ETag** header is fully supported, along with `If-None-Match`.

How does NGINX handle byte-range requests?

If the file is up-to-date in the cache, then NGINX honors a byte-range request and serves only the specified bytes of the item to the client. If the file is not cached, or if it's stale, NGINX downloads the entire file from the origin server. If the request is for a single byte range, NGINX sends that range to the client as soon as it is encountered in the download stream. If the request specifies multiple byte ranges within the same file, NGINX delivers the entire file to the client when the download completes.

Once the download completes, NGINX moves the entire resource into the cache so that all future byte-range requests, whether for a single range or multiple ranges, are satisfied immediately from the cache.

Please note that the upstream server must support byte-range requests for NGINX to honor byte-range requests to that upstream server.

Does NGINX support cache purging?

NGINX Plus supports selective purging of cached files. This is useful if a file has been updated on the origin server but is still valid in the NGINX Plus cache (the `Cache-Control:max-age` is still valid, and the timeout set by the `inactive` parameter to the `proxy_cache_path` directive has not expired). With the cache-purge feature of NGINX Plus, this file can easily be deleted.

How does NGINX handle the Pragma header?

The `Pragma:no-cache` header is added by clients to bypass all intermediary caches and go straight to the origin server for the requested content. NGINX does not honor the Pragma header by default, but you can configure the feature with the following `proxy_cache_bypass` directive:

```
location /images/ {
   proxy_cache my_cache;
   proxy_cache_bypass $http_pragma;
   # ...
}
```


www.dbooks.org

Further Reading

There are many more ways you can customize and tune NGINX caching. To learn even more about caching with NGINX, please take a look at the following resources:

- The `ngx_http_proxy_module` section of the of the NGINX documentation contains all of the configuration options for content caching.
- The NGINX Content Caching webinar is available on demand. The webinar steps through much of the information presented in this blog post.
- The Content Caching section of the NGINX Plus Admin Guide has more configuration examples and information on tuning the NGINX cache.
- The Content Caching with NGINX Plus product page contains an overview on how to configure cache purging with NGINX Plus and provides other examples of cache customization.

More than 50% of the world's busiest websites use NGINX, significantly for its web-acceleration and content-caching capabilities. For more solutions and more details, check out the blogs and the feature briefs at nginx.com, which speak to capabilities in NGINX and NGINX Plus. And take a look at our webinar list.

And if you'd like to investigate these capabilities further, of course there's the documentation and solutions that you can find at nginx.org and nginx.com, but nothing beats taking a download and trying it out. The open source product can be found at nginx.org, or try the supported product with additional load balancing, application delivery, management, and ease-of-use features at nginx.com.

Selected Website Links:

1. https://www.nginx.com/blog/nginx-high-performance-caching/
2. https://www.nginx.com/blog/nginx-caching-guide/
3. https://www.nginx.com/blog/maximizing-php-7-performance-with-nginx-part-i-web-serving-and-caching/
4. https://www.nginx.com/blog/maximizing-drupal-8-performance-nginx-part-ii-caching-load-balancing/
5. https://www.nginx.com/blog/cache-placement-strategies-nginx-plus/
6. https://www.nginx.com/blog/benefits-of-microcaching-nginx/
7. https://www.nginx.com/blog/smart-efficient-byte-range-caching-nginx/
8. https://www.nginx.com/blog/shared-caches-nginx-plus-cache-clusters-part-1/
9. https://www.nginx.com/blog/shared-caches-nginx-plus-cache-clusters-part-2/

www.dbooks.org