FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Sharing HTTP cache summaries between web servers
Student:	Ondřej Polanecký
Supervisor:	Ing. Daniel Sedlák
Study program:	Informatics
Branch / specialization:	Software Engineering 2021
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2024/2025

Instructions

Nginx is an advanced web server primarily employed as a reverse HTTP proxy. Widely adopted by numerous Content Delivery Networks (CDNs) for caching, Nginx frequently operates in multilayer setups within CDNs to enhance the cache hit ratio. In the real world, when a client requests cachable data, the request goes from the lower-level cache to the higher-level cache until the cachable data is requested from the origin.

The core objective of this thesis is to further enhance the cache hit ratio by implementing a mechanism for sharing cache summaries from upper-level caches to lower-level caches. This process will enable lower-level caches to prioritize accessing the higher-level cache, which may already contain the desired file in its cache.

Additionally, this thesis seeks to examine diverse methodologies for creating cache summaries and transferring these summaries from upper cache layers to lower cache layers. The proposed solution must consider both used memory and network capacity.

Optionally, the thesis can explore the development of a viable solution tailored for scenarios involving cache eviction.

1. Get familiar with the current multilayer HTTP cache setup consisting of Nginx web servers (infrastructure provided by the supervisor).
2. Research various bloom filter types that could be used as cache summaries and find the most optimal one in terms of available memory and network bandwidth.

Electronically approved by Ing. Michal Valenta, Ph.D. on 23 December 2023 in Prague.

3. Implement suitable cache summary and exchange algorithms based on previous research. Optionally implement a solution tailored for scenarios involving cache eviction.
4. Evaluate and compare the following metrics before and after deyploying cache exchange summaries.
   - Cache HIT/MISS ratio
   - IN/OUT/interlayer traffic
   - Memory usage

Electronically approved by Ing. Michal Valenta, Ph.D. on 23 December 2023 in Prague.

Bachelor's thesis

SHARING HTTP CACHE SUMMARIES BETWEEN WEB SERVERS

Ondřej Polanecký

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Daniel Sedlák
May 16, 2024

Czech Technical University in Prague
Faculty of Information Technology
© 2024 Ondřej Polanecký. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Polanecký Ondřej. *Sharing HTTP cache summaries between web servers*. Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

Acknowledgments vi

Declaration vii

Abstract viii

List of abbreviations ix

Introduction 1

1 About this thesis 2
- 1.1 Reverse proxy 2
- 1.2 Scalability of reverse proxy with caching 3
  - 1.2.1 Consistent hashing 4
- 1.3 Multilayer cache 4
- 1.4 Exact problem formulation 5
- 1.5 Thesis structure 6

2 Existing solutions 7
- 2.1 Per-request query 7
- 2.2 Summary cache 7
- 2.3 Squid 8

3 Research of Bloom filters 9
- 3.1 Algorithm description 9
  - 3.1.1 Deciding the parameters 10
- 3.2 Use cases 11
- 3.3 Types of Bloom filters 12
  - 3.3.1 Counting Bloom filter 12
    - 3.3.1.1 Counter overflow 12
  - 3.3.2 Sliding Bloom filter 13
    - 3.3.2.1 Improved sliding Bloom filter 13
  - 3.3.3 Scalable Bloom filter 14
  - 3.3.4 Blocked and register Bloom filter 14
    - 3.3.4.1 Register Bloom filter 15

4 Implementation of Bloom filters in NGINX 16
- 4.1 NGINX 16
  - 4.1.1 Shared memory 17
  - 4.1.2 HTTP file cache 18
- 4.2 Bloom filter implementation 19
- 4.3 Bloom filter module 22
  - 4.3.1 Allocations in shared memory 22
- 4.4 New directives 24

5 Design and implementation of summary sharing **26**
5.1 lua-nginx-module 26
5.1.1 Directives 26
5.1.2 FFI 27
5.1.3 Worker synchronization 28
5.2 Bloom filter sharing 29
5.2.1 Downloading the Bloom filters 29
5.2.1.1 Frequency of downloading 31
5.3 Selecting the upstream server 31

6 Testing and evaluation **33**
6.1 What went wrong 35
6.1.1 Downloading of Bloom filter 35
6.1.2 Shared dictionary 35
6.2 Evaluation 35

7 Conclusion **38**

The contents of the included media **41**

List of Figures

1.1 Sharded cache setup.[5] 4
1.2 Schema of multilayer setup. 5
1.3 Checking that content is in the cache of neighboring servers. 6

3.1 False positive probability function 11
3.2 Filter evolution when adding elements and shifting. Each letter (A,B,. . .) represents a generation of g elements. With $k = 3$ and $l = 2$ [14] 13
3.3 Checking of blocked Bloom filter using bitmask 15

4.1 NGINX shared memory overview[20] 24

5.1 OpenResty directives and their phases.[23] 27

6.1 Regular pattern in our Bloom filter that indicates something is wrong. 34
6.2 Fixed Bloom filter without any irregularities. 34
6.3 Traffic that was served from cache on current server or from neighboring cache. . 36
6.4 Percentage of requests, that were not served from any cache. 36

List of Tables

List of code listings

4.1 Minimal caching reverse proxy configuration. 17
4.2 Definition of NGINX struct for shared memory.[17] 17
4.3 Definition of file cache struct in shared memory.[17] 18
4.4 Definition of NGINX struct representing cache entry. 20
4.5 Definition of Bloom filter structs. 21
4.6 Updating the counting Bloom filter. 21
4.7 Function to get hash values from cache entry key. 22
4.8 Adding shared memory zone for Bloom filter.[18] 23
4.9 Nginx config with Bloom filters. 25

5.1 Compare-and-swap function for shared dictionary. 30
5.2 Location block for downloading Bloom filter into shared memory. 31
5.3 Downloading the Bloom filter from Lua. 31

I would like to express my deepest gratitude to my friends, whose unwavering support and encouragement have been invaluable throughout this journey. Their patience, understanding, and occasional distractions have provided me with the motivation and strength to persevere through the challenges of thesis writing.
I am immensely grateful to my supervisor, Ing. Daniel Sedlák, for their guidance, expertise, and constant encouragement. Their insightful feedback, constructive criticism, and dedication to my academic growth have been instrumental in shaping this thesis into its final form.
Special thanks are also due to my opponent, RNDr. Jan Prachař, whose thoughtful insights and constructive critiques have significantly contributed to the implementation of the solution presented in this thesis. Their expertise and willingness to engage in rigorous academic discourse have enriched the quality of my work and challenged me to strive for excellence.
I am indebted to all those who have supported and believed in me throughout this academic endeavor. Your contributions have played an integral role in the completion of this thesis, and for that, I am profoundly grateful.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. I further declare that I have concluded an agreement with the Czech Technical University in Prague, on the basis of which the Czech Technical University in Prague has waived the right to conclude a licence agreement on the utilization of this thesis as a school work pursuant to Section 60(1) of the Copyright Act. This fact does not affect the provisions of Section 47b of the Act No. 111/1998 Coll., on Higher Education Act, as amended.

In Praze on May 16, 2024

Abstract

Caching is one of the most effective methods for speeding up most applications. It is therefore important that it is used effectively. In this paper, we will focus on improving the cache utilization of the NGINX web server when deployed in multilayer setup. We will mention how NGINX works when deployed in a multilayer setup and what are the advantages of this setup. We then present a way to share the cache between multiple web servers in this deployment and thus improve cache utilization. Currently, this solution is not widely used, and most widespread webservers do not have this functionality. Therefore, this thesis will also focus on the implementation of this functionality in the NGINX web server. The Bloom filter data structure is also part of this implementation, so we will discuss it more in the research part. Also, we will then test and deploy the solution into the production environment, to get its effectivity in real-world scenarios.

Keywords Bloom filter, NGINX, NGINX Lua module, CDN, HTTP cache, Lua, C, Summary cache, Cache sharing

Abstrakt

Cachování je jedna z nejefektivnějších metod na zrychlení většiny aplikací. Je tedy důležité, aby byla cache co nejvíce využívána. V této práci se budeme věnovat zlepšení využití cache webového serveru NGINX ve vícevrstvém prostředí. Zmíníme jak funguje NGINX při nasazení ve více vrstvách a jaké má toto nasazení výhody. Poté představíme způsob jak v tomto nasazení sdílet cache mezi více webovými servery a tudíž zlepšit její využití. Momentálně toto řešení není příliš rozšířené a v NGINXu i jiných populárních serverech tato funkcionalita chybí. Tato bakalářská práce se tedy zaměří i na implementaci této funcionality do webového serveru NGINX. Součástí tohoto řešení je i datová struktura Bloom filter, takže se jí v rešeršní části budeme více věnovat. Následně tuto implementaci otestujeme a nasadíme do produkčního prostředí, abychom zjistili její efektivitu v reálných scénářích.

Klíčová slova Bloom filter, NGINX, NGINX Lua module, CDN, HTTP cache, Lua, C, Summary cache, Sdílení cache

List of abbreviations

CDN	Content Delivery Network
FFI	Foreign Function Interface
FPGA	Field-programmable gate array
FPR	False positive rate
HTCP	Hyper Text Caching Protocol
HTTP	Hypertext Transfer Protocol
ICP	Internet Cache Protocol
LRU	Least Recently Used
SIMD	Single Instruction, Multiple Data
TTL	Time to live
URI	Uniform Resource Identifier
URL	Uniform Resource Locator

Introduction

Caching is one of the most important techniques to improve the performance of almost any application. Its primary objective is to store frequently accessed data, allowing faster retrieval and consequently reduce the load on underlying systems. There are many types of caches, but in this thesis, we will focus on content caching in the context of web servers.

The main aim of this bachelor's thesis is to improve the cache efficiency of NGINX[1] webserver in a multilayer setup. First, we will introduce the concept of multilayer HTTP cache, explain why it is beneficial and propose further improvement by sharing the cache. Specifically, we need to solve the problem of sharing cache between multiple caching reverse proxies. We will explore the existing solutions and their shortcomings. To implement the solution, we will need to study in detail the Bloom filter data structure, because it will be the core of our solution and it is required in the assignment. With the acquired knowledge, we will implement Bloom filters into NGINX. Lastly, we will dive into the internals of NGINX and its Lua module to implement the mechanism for sharing cache between servers. Then we will test the solution test and evaluate the solution in testing environment. At last we will test the solution in a real-world scenario and evaluate important metrics.

Chapter 1
About this thesis

The first chapter of this thesis will introduce the thesis itself. It will explain what a multilayer HTTP cache is, its use case, and the problem it aims to solve. Then we will propose a way to further improve the cache efficiency of NGINX in a multilayer. Additionally, the structure of the thesis and the topics covered in each chapter will be outlined. This serves as an extension of the introduction chapter to ensure a clear understanding of the problem and what to expect from the rest of the thesis.

First, I will explain some basic terms and concepts that are necessary to understand the problem we are trying to solve.

1.1 Reverse proxy

Multilayer HTTP cache consists of multiple caching reverse proxies. The role of a reverse proxy is to forward client requests to the relevant backend servers and then send the backend's response back to the client. From the client's viewpoint, the reverse proxy operates as the origin server. Origin server is the server the source of the content. It does not do any additional proxying.

Reverse proxy can also provide these additional features:

- **Load balancing -** We can have multiple backend servers, and the reverse proxy can distribute the load between them.
- **Security -** Because backend servers are hidden behind the reverse proxy, they cannot be so easily directly targeted by attacks like DDOS. Reverse proxy can also provide some security features like rate limiting, WAF, etc.
- **Caching -** Storing the response from the backend servers and returning it to the client. This can greatly reduce the load on the backend servers and improve the response time. This functionality is our main focus in this thesis.
- **Additional functionality -** Reverse proxy can also provide additional functionality like SSL encryption, compression, or even something like rewriting the response.

The flow of request when using a reverse proxy is as follows. First, the client sends a DNS request for the domain name it wants to access. Then the DNS instead of returning the IP address of the origin server, returns the IP address of some reverse proxy. The client then sends a request to this reverse proxy. The reverse proxy checks if the requested content is in its cache and if it is valid (content can be expired, accessible only to certain users, etc.). If it is, it returns the content to the client right away. If not, it will forward the request to the origin server and

return the response from the origin server. Depending on the Cache-Control header[2] or custom proxy configuration, the reverse proxy may cache the response from the origin server. This means that any subsequent requests for the same content can be served from the cache without having to go to the origin server.

1.2 Scalability of reverse proxy with caching

Suppose we need to make our reverse proxy more fault-tolerant and scalable. For scaling, we can at first just add more memory, CPU, storage, etc. But at some point, we could reach the limit of what one server can handle. On top of that, if we have just one server, it is a single point of failure.

So naturally, we want to be able to scale our reverse proxies horizontally. Scaling horizontally means adding more instances of the reverse proxy. As opposed to vertical scaling, which means adding more resources to the existing server.

Now if we only added more reverse proxies without any additional configuration and let clients randomly choose which reverse proxy to use. We will have to face some problems. A lot of requests for the same content will be sent to different reverse proxies, so the content will not be cached effectively. The same files would be stored in multiple caches, so we would waste memory. We would also be forced to forward the request to the origin more than necessary because we could be requesting some file that is already in the cache of another reverse proxy.

One way to solve the scalability and fault tolerance is to shard the cache. Before the caching proxies, we add some number (depending on the desired fault-tolerance) of load balancers, that will decide to which caching proxy the request will be forwarded. Load balancers can have a lot of different strategies to decide which server to forward the request. To name a few:[3]

- Round-robin - Forward the request every time to the next server in the list. (Cycle through the list)
- Least connections - Forward the request to the server with the least number of active connections.
- IP hash - Forward the request to the server based on the hash of the client's IP address. Useful for session persistence.

It is also possible for all of these methods to set weights for servers, so some servers will get proportionally more requests. For sharding of the cache, we need to use load balancer with consistent hashing. That will guarantee that for the same file, every load balancer will forward the request to the same caching proxy.[4] And in case we add or remove another caching proxy, only a fraction of the files will be remapped to a different caching proxy. More on consistent hashing in the next subsection.

The final setup can look something like Fig. 1.1.

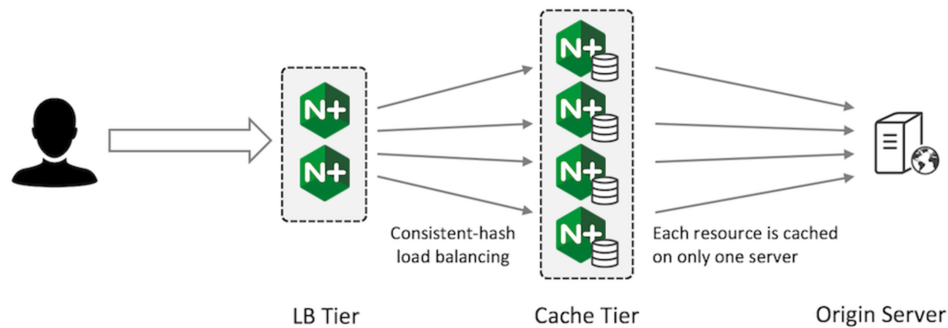


Figure 1.1 Sharded cache setup.[5]

1.2.1 Consistent hashing

Consistent hashing is a technique used to map keys to servers in a way that minimizes the number of keys that need to be remapped when a server is added or removed. It is widely used in distributed systems.[6] Common use cases are distributed databases, load balancing, or caching.

The abstract idea is that we have a hash function that maps keys to a circle. Each server is then represented by a certain number of points on the circle. (We can prioritize servers by adding more points for them.) To locate a server for a key, we use the hash function to map the key to a point on the circle and then move clockwise to find the first server point. This is the server that will be returned for the key.

This way, when we add (or remove) a server to the system, only a fraction of the keys will be remapped to a different server. When we add a server, no key will be remapped between the old servers. Every remapped key will be mapped to the new server. And because every server should on average take up $1/n$ of the circle, where n is the number of servers. The portion of keys that need to be remapped is also on average $1/n$.[4]

If we used just some regular hash function with modulo to map keys to servers, Almost everything would have to be remapped when we add or remove a server. (Because the modulo would change.) On the other hand, a regular hashing table would be faster. Consistent hashing has time complexity $O(\log n)$ for lookup, and regular hashing with modulo has time complexity $O(1)$. Therefore, consistent hashing should only be used when the number of servers is expected to change, and reassigning keys could be expensive. For instance, in distributed storage, reassigning keys after adding servers would require moving almost all the data between servers.

1.3 Multilayer cache

Now we can finally talk about the multilayer HTTP cache setup.

The idea is to have multiple layers of caching proxies. Each layer has a different threshold for caching content. The threshold for caching is the number of requests for the same content after which the content will be cached. The first layer has the highest threshold, so it caches only the most popular content. The last layer has the lowest threshold (typically 1), so it caches everything else that the lower layers did not cache. This setup can solve the problem if we have a lot of content that is requested only a few times, but it adds up to a lot of traffic. If we had just one layer with a constant threshold, we would have to choose between having the threshold too low and caching things that may never be requested again or having the threshold too high, and a lot of traffic would not be cached at all.

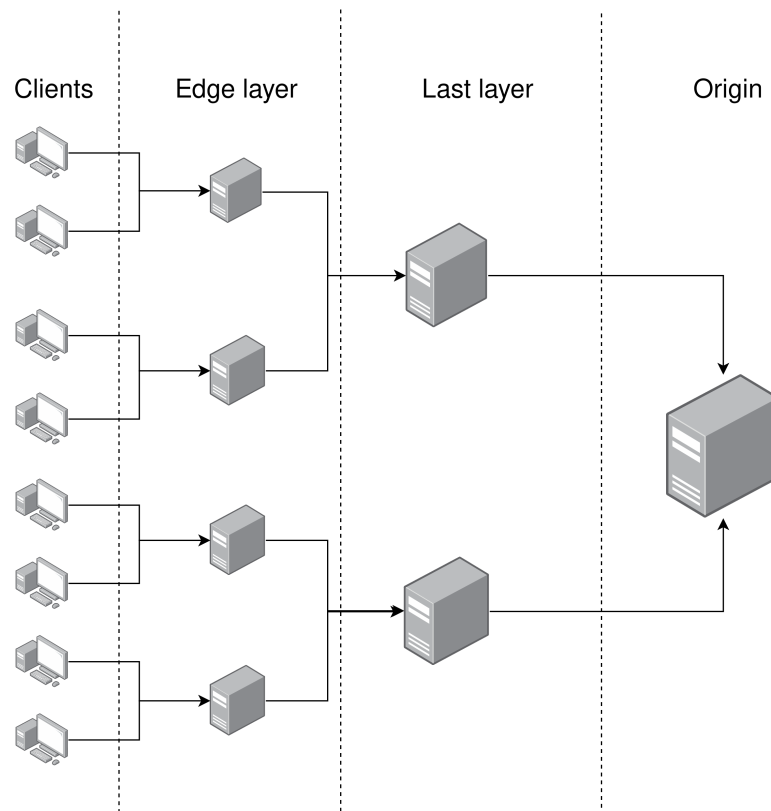


Figure 1.2 Schema of multilayer setup.

With sufficiently large infrastructure we can set the layers thresholds so that the cache will converge to storing a majority of the content. That is good because the origin server will not be overloaded with requests and it also reduces potential egress costs[1].

The first layer is the edge layer. This layer consists of caching proxies that are closest to the clients. Also, this layer is the only layer communicating with the clients. It is geographically distributed, and clients will always connect to the closest server.[2]

When each layer does not have the content in its cache, it will forward the request to the next layer. Every server has specified the closest server in the next layer where it should forward the request. Servers can have sharded cache so we have to use consistent hashing to determine to which shard of the server we should forward the request.

With this setup, the more the content is popular the closer it will be to clients. Also depending on how large the infrastructure is, you can set the thresholds so that the cache will converge to storing a majority of the origin server content.

In Figure 1.2 we can see the schema of the multilayer cache setup.

1.4 Exact problem formulation

The problem we are trying to solve is that the cache/hit ratio could still be improved.

For example, when a request reaches the last layer and the content is not present in the cache, it could potentially be present in the cache of a different server on the same layer (or any other server that it did not go trough). Unfortunately, we do not have this information, so we end up

[1]Fees from cloud providers for transfering data from storage.
[2]This can be done using DNS servers that will return the IP address of the closest server.

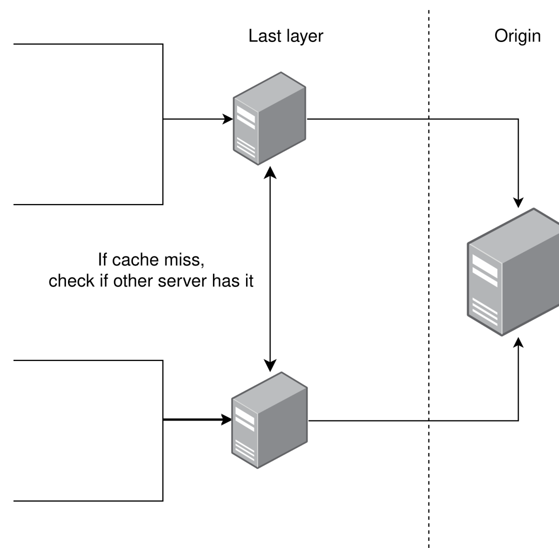


Figure 1.3 Checking that content is in the cache of neighboring servers.

forwarding the request to the origin server, causing an unnecessary load on it. This is where we need to come up with a mechanism to check if the content is cached in any of the neighboring servers on the same layer. If we can find the content in one of these servers, we can forward the request to that server and return the content to the client.

Therefore, the exact problem we need to solve is to create a mechanism that can help us identify which neighboring server contains the requested content. As demonstrated in Figure 1.3.

1.5 Thesis structure

We will briefly describe the structure of the thesis and what each chapter will cover. In the next chapter 2, we will explore the existing solutions to the problem of sharing cache between multiple caching reverse proxies. We will explain why these solutions are not ideal for our use case and propose a new solution. In the third chapter 3, we will cover the research of Bloom filters. Some of their variants and what problem each of them solves. In the chapter 4 we will focus on the design and implementation of Bloom filters into NGINX. After that the chapter 5 will be focused on implementing the mechanism for sharing cache between servers. In the chapter 6, we will look at our implementation in a real-world scenario. We will evaluate metrics like cache hit ratio, memory usage, and bandwidth usage. Then compare them to values before deploying the solution. Also, we will talk about testing the solution and what went wrong during implementation. Finally in the last chapter 7, we will summarize the thesis and our results and achievements. Also, we will talk about possible future work and improvements.

Chapter 2
Existing solutions

In this chapter, we will explore the existing solutions and their shortcomings.

2.1 Per-request query

One solution is to query all the neighboring servers to check if the content is in their cache for each request. This has som drawbacks:

- Increased latency - We have to wait for the response of the query.
- Increased load on the servers - By sending additional request to all the servers on every cache miss.
- It is not scalable - We can see that by increasing the number of servers, the number of queries grows quadratically. If any server has the content, we need response only from this server. But if none of the servers have the content, we need to wait for all the responses.

This solution is not widely used because of the increased latency and network overhead. [7] There are few different protocols (ICP, HTCP etc.) for using this solution, but they differ only in the way how optimized the queries are. None of them solves the fundamental drawbacks of this approach.

2.2 Summary cache

A way more efficient solution is to use a summary cache protocol.[7]

Summaries in this context mean some kind of data structure that will allow us to quickly check if a specified string (in most cases URL) is in it. Each server has a summary of the cached content for all the servers it needs to forward the request to. When the server receives a request, it first checks all the summaries of the neighboring servers. Typically, this is implemented using Bloom filters (details in the chapter 3) because they are space-efficient and sufficiently fast. If the content is in the summary, it forwards the request to that server. Every proxy has to produce a summary of its cache and periodically send it to all the other proxies. This has the same drawback of $O(n^2)$ sent messages, but in this case it is significantly less problematic, because we do not send it on every request. The servers then periodically exchange the summaries. That means that the summary are not always up to date, but that is a trade-off we have to make. This way, we can avoid the main drawbacks of the per-request query solution. The latency is reduced because we do not have to wait for the response of the query requests. Also, the inter-server

messages are greatly reduced. The paper that proposed this solution claims that the inter-server messages are reduced by a factor of 25–60, the bandwidth usage is reduced by 50 % and maintains almost the same HIT ratio as the per-request query solution.[7]

The bad thing about this solution is that there is not many usable implementations of it. The only web server that I know of that has some kind of support for this is Squid.

2.3 Squid

Squid is a caching and forwarding HTTP web proxy that has implemented Cache Digests.[8] (a variant of summary cache) However, we need to implement our solution into NGINX so let us look at how does Squid do it.

Squid uses a standard Bloom filter for representing summaries. This means that it cannot perform deletions from the summaries, so it does not support cache purges and cache eviction. It has to periodically completely rebuild the whole filter, so it stays relevant (default is every 1 hour). The rebuild can be an expensive operation depending on the number of cache entries. Every time the capacity of the cache is increased by more than 10 % it also increases the size of the Bloom filter in the next rebuild. [9]

In the next chapter, we will cover the research of Bloom filters because it is necessary for the correct and efficient implementation of the solution.

Chapter 3

Research of Bloom filters

Bloom filter is a probabilistic data structure used to test whether an element is a member of a set. It is space-efficient and fast, but the tradeoff for small memory usage is that it can produce false positives. False negatives are not possible. Bloom filter does not store the actual elements, they are hashed into the filter. As a result, it is not possible to retrieve the elements that have been inserted. The only function of a Bloom filter is to check whether an element might have been inserted into it.

The standard Bloom filter has two parameters: m and k.

- m is the size of the bit array.
- k is the number of hash functions.

  Each hash function should uniformly generate values in the range from 0 to $m-1$ and be independent of the other hash functions.

The time complexity for insertion and query is $O(k)$. That means it is constant to the number of inserted elements. Also, for most practical purposes, k is a small number.

The probability of false positives can be estimated based on these parameters and the number of elements inserted into the filter. We will cover this in more detail in section 3.1.

There are many types of Bloom filters, such as counting Bloom filter, scalable Bloom filter, blocked Bloom filter, etc. For now, we will focus on the standard Bloom filter and touch on other types later.

3.1 Algorithm description

We start with an empty bit array (all bits are set to 0). To insert an element into the filter, we need to compute k hash values for the element. Each hash value should be modulo m, so it is in the range from 0 to $m-1$. Practically, we use one hash function that generates a hash of at least $m*k$ bits and then split it into k parts. Then we set the bits at the computed indices to 1.

The hash functions should generate random uniformly distributed values. They should also be independent of each other. It is important to note that removing an element from the filter is not possible. For this purpose, we would need a counting Bloom filter (more on that in subsection 3.3.1).[1]

To check if an element is in the filter, we compute hash values the same way as when inserted. If all the bits at the computed indices are 1, the element may be in the filter. If not, the element

[1] There are also more advanced types of Bloom filters that allow removing elements, such as the d-left counting Bloom filter.[10] We will not cover them in this thesis.

is not in the filter. The probability of an element not being in the filter if all its bits are 1 is called the false positive probability.

The probability of a false positive query in an arbitrary Bloom filter can be estimated by this formula:[11]

$$(1-(1-\frac{1}{m})^{kn})^k$$

, where n is the number of elements inserted into the filter.

- $1-\frac{1}{m}$ is the probability that a random bit is 0.
- $(1-\frac{1}{m})^k$ if hash functions are independent, this is the probability that a random bit is set to 0.
- $((1-\frac{1}{m})^k)^n$ this is the probability that bit is set to zero after inserting n elements.
- $1-(1-\frac{1}{m})^{kn}$ if we reverse the previous probability, we get the probability that a random bit is set to 1.
- $(1-(1-\frac{1}{m})^{kn})^k$ finally, we compute the probability that all k hash functions return indexes of bits that are set to 1.

This is equivalent to the probability that the query is a false positive.

We computed the probability of false positives in an arbitrary Bloom filter.

We can also compute the probability of a false positive query in a specific Bloom filter. Inserting n elements into the filter can result each time in a different number of bits set to 1. (In the extreme case, a bad hash function could map everything to one index) So there is some variance.

So to calculate the probability of a false positive query in a specific Bloom filter, we can use this formula:

$$\left(\frac{s}{m}\right)^k$$

Where s is the number of bits set to 1.

That is because we have an array of m bits and s of them are 1. Query is a false positive if all hash functions return indexes of a bit that is set to one. The Probability that random index is set to 1 is $\frac{s}{m}$. To get a probability that all k hash functions return indexes of bits that are set to 1, we can multiply them together k times. This is possible because of the independence of the hash functions.

3.1.1 Deciding the parameters

We need to put some thought into choosing the parameters m and k, because it cannot be changed after the filter is created. Rebuilds are not easily possible because the filter does not store the actual elements. We would need to have a separate list of elements that were inserted into the filter.

So we have to balance our available memory, performance cost of more hash functions and desired probability of false positives.

So typically, we know approximately how many elements will be inserted into the filter and how much memory we can afford to use. To determine the ideal number of hash functions k, we need to minimize the false positive probability function with respect to k.

For simplicity, we can use the approximation of the false positive probability function:[11]

$$\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^k \approx (1-e^{kn/m})^k$$

And use derivations to find the minimum of this function.

Which is approximately at $k = \frac{m}{n} \ln 2$. We can use a similar approach to find the optimal values of other parameters. It is important to note that the false positive rises exponentially with the number of elements inserted into the filter, so generally we want to rather overestimate the number of inserted elements. In Fig. 3.1 a simple plot of the false positive probability function if we choose $m = 1$ MB and $k = 4$ The red line is false positive rate of 1 in 100 and the green line is false positive rate of 1 in 1000.

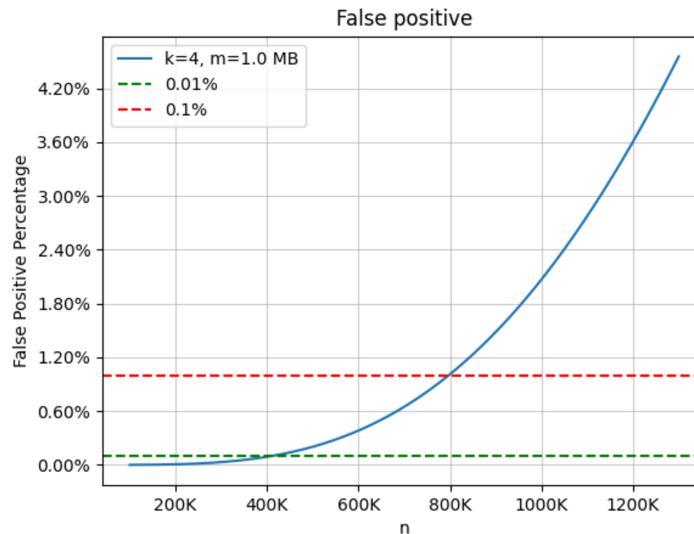


Figure 3.1 False positive probability function

The bits per element for a Bloom filter that has the optimal k can be estimated by the desired probability of false positives:[12]

$$c = -1.44 * log2(p)$$

Where c is the bits per element and p is the probability of false positives. For a given p it is not possible construct a Bloom filter with fewer bits per element than this estimate.

3.2 Use cases

Bloom filters are used in many different areas, where we need to quickly perform a membership test with allowed false positives.

Some of the usual use cases are:

- Database - check if key is in the database before querying the database.
- Browser - Bloom filter containing malicious URLs to check if the URL is safe. Positive result triggers a more expensive check.
- Monitoring network traffic - Bloom filters can be used in deep packet inspection. For example detecting if the packet contains a specific string in the payload. On hardware devices (routers), they could also be implemented as FPGA.

Bloom filters are not used in cases where false positives are not acceptable, or in low-throughput applications in which the accuracy is more important than speed and memory usage.

3.3 Types of Bloom filters

The standard Bloom filter has some limitations.
To name a few:

- Not possible to remove elements.
- Not possible to count the number of occurrences of an element. (We can insert the same element multiple times, but we cannot count it.)
- Bad cache locality. For large Bloom filters, almost all lookups will result in a cache miss.

And for these limitations, there are some alternatives to the standard Bloom filter. I will mention some of them. This, however, is not an exhaustive list. There are a lot of Bloom filter variations. The research on Bloom filters is still ongoing, and new types are being developed.[13] These filters were selected because they solve a specific problem of the standard Bloom filter in a simple way.

3.3.1 Counting Bloom filter

The counting Bloom filter is a simple variation of the standard Bloom filter that allows us to remove elements from the filter. The idea is that if we want to do removal in the standard Bloom filter, it is possible that we would remove a bit that is shared with another element. So for counting Bloom filter, we replace the bit array with an array of counters to know if the bit is shared with another element. Every bit is now represented by a counter. Instead of setting the bit to 1, we increment the counter. And when we want to remove an element, we decrement the counter.

Then, when we want to check for the presence of an element, we check if all the counters are greater than 0. The counting filter is also trivially convertible to the standard Bloom filter.

One of the drawbacks of the counting Bloom filter is that it is not as space-efficient as the standard Bloom filter. It will take n times more memory, where n is the size of the counter in bits.

3.3.1.1 Counter overflow

For most practical purposes, the counter of size 4 bits is more than enough. But there is always a possibility that the counter will overflow. In case of counter overflow, because it will set a non-zero index to zero, it will cause false negatives (query for existing elements can be negative). So we will lose one of the main advantages of the Bloom filter.

The probability that any counter in the Bloom filter is greater or equal to i can be estimated as:

$$P(\max(c) >= i) <= m \binom{nk}{i} \frac{1}{m^i} <= m \left(\frac{enk}{im} \right)^i$$

In 3.1.1 we estimated that the optimal value of k is $\frac{m}{n} \ln 2$. So we will assume that $k = \frac{m}{n} \ln 2$.

$$P(\max(c) >= i) <= m \left(\frac{e \ln 2}{i} \right)^i$$

Now we can simply plug in the values of m, n, and i to get the probability of counter overflow. The four bit counter will overflow on count 16 and higher so $i = 16$.

$$P(\max(c) >= 16) <= 1.37 \times 10^{-15} * m$$

[7]

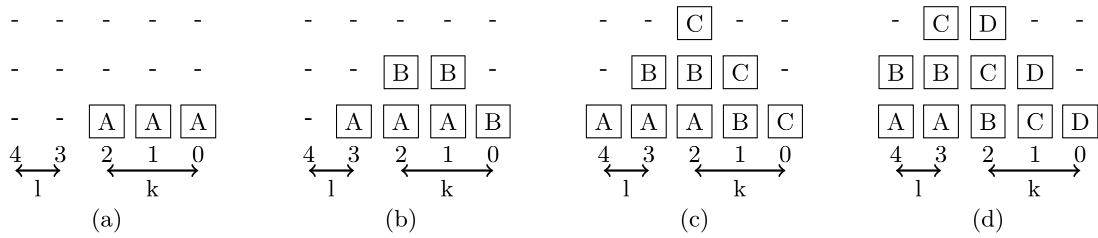


Figure 3.2 Filter evolution when adding elements and shifting. Each letter (A,B,...) represents a generation of g elements. With $k = 3$ and $l = 2$ [14]

The probability of any overflow will reach 1 percent at $m \approx 912GB$. This is not a typical size of Bloom filters, we do not have to worry about counter overflow in most cases.

We would probably have to extremely overfill the filter to reach any significant probability of overflow. And by then, the filter would probably be useless anyway.

The Only realistic scenario where we could reach the overflow is that if we used the filter also for counting the number of occurrences of elements. And inserting the same element multiple times.

3.3.2 Sliding Bloom filter

Sliding Bloom filter is a very simple method to automatically expire old elements from the Bloom filter. The use case of this filter could be stream processing or when there is a data retention policy. For example, we could use a sliding Bloom filter in monitoring, to always have a filter with at least the last 24 hours of seen IP addresses.

The basic idea is that we have $n >= 2$ Bloom filters. We have them organized in a sort of sliding window. The newest filter is the master filter. Every insertion operation is done on the master filter. After a specified time or number of insertions, the oldest filter is discarded and a new empty master filter is created. (We will reuse the discarded filter)

When we want to check if an element is in the filter, we check all the filters. This is one of the main drawbacks of this method. Apart from the fact that we have to query more Bloom filters. The probability of false positives is exponentially rising with the number of filters. When we have n filters, the probability of false positives is p^n where p is the probability of false positives in one filter. For some use cases, this could be acceptable if we have a low number of filters.

3.3.2.1 Improved sliding Bloom filter

Because of the drawbacks of the sliding Bloom filter, there is an improved version of it. It is called age partitioned Bloom filter.

The idea is that we have k Bloom filters that are being used for insertion and l Bloom filters that are stored only for querying. When we want to insert an element, we insert it into all k filters.[14] This is the downside of this method, that we will have more duplicated data and slower insertion speed.

The Bloom filters move in time like this Fig. 3.2. In (d) when the first Bloom filter that contained element A is discarded, it is no longer considered to be present in the filter. (There are not 3 consecutive Bloom filters that contain element A.)

When we want to check if an element is in the filter, we have to check if there are at least k consecutive filters that contain the element. This solves the problem of exponentially rising false positives with the number of filters. Because to have a false positive, it would need to be a false

positive in all of the k filters as opposed to the basic version where only one filter needs to be a false positive.

Expiration works similarly to the basic sliding Bloom filter. We have some heuristic that decides when to discard the oldest filter. When we discard the oldest filter (the one on the left)

3.3.3 Scalable Bloom filter

This type of Bloom filter eliminates the need to know the approximate maximum number of elements inserted into the filter. It will dynamically scale the Bloom filter size to infinity without rebuilding. If implemented correctly, the false positive probability will converge to some wanted value. However, for optimal performance and space efficiency, it is still good to have a rough estimate of the number of elements.

This method of scaling Bloom filters is kind of similar to resizeable arrays. We start with some with a small Bloom filter.

When it reaches some threshold, we create a new Bloom filter with s times more bits. s is the scaling factor. After creating a new filter, all insertions are done on the new filter. This process can be repeated indefinitely. However, depending on the desired false positive probability and scaling factor, the number of filters and memory usage can grow very quickly.

To check if an element is in the filter, we have to check all of the filters. This is the same problem as with the sliding Bloom filter. The probability of checking multiple filters would be exponentially rising with the number of filters. But here it is solved by using different maximum false positive probabilities for each filter. To easily guarantee that the probability of false positives will converge to some wanted value, we have to satisfy this recursive formula:

$$p_{i+1} = p_i * r$$

Where p_i is the probability of false positives in the i-th filter. And $0 < r < 1$ is the tightening ratio. [15]

The main advantage of this method is that we could stress-free use way smaller Bloom filters with the option to scale them up if needed. With using statically sized Bloom filters, you most of the time have to grossly overestimate the number of elements. That is because there is not the option to resize the filter apart from rebuilding it and the false positive probability would rise exponentially with the number of elements.

3.3.4 Blocked and register Bloom filter

Bloom filters are generally fast enough for most use cases. However, there are some use cases where we need even faster lookups. Mostly in big data processing. One example where Bloom filters were not fast enough is this Cloudflare usecase[2]. They tried to remove duplicates from a very large file with IP addresses. But when benchmarking, they found out that the classic hash table with linear probing was approximately 5 times faster than a Bloom filter.

They did profile the program and the Bloom filter code spent the majority of time waiting for the memory to be read. The reason for this is that large Bloom filters have bad cache locality. When we have a Bloom filter that is way larger than any cache in the CPU, almost every lookup will result in a cache miss. Because in one query, we have to perform k lookups in the bit array and every lookup should be in a random place in the array if the hash function is sufficiently random. So any cache prefetching is useless. Also, as we know, optimal k increases with the size of the filter, so the problem is even worse for larger filters. This would not be such a problem when using very small Bloom filters, because they would fit at least into the L3 cache of processor.

This problem can be solved by using blocked Bloom filters. The obvious way to solve cache misses is to increase the locality of the data. We do this by dividing the bit array into blocks.

[2] https://blog.cloudflare.com/when-bloom-filters-dont-bloom

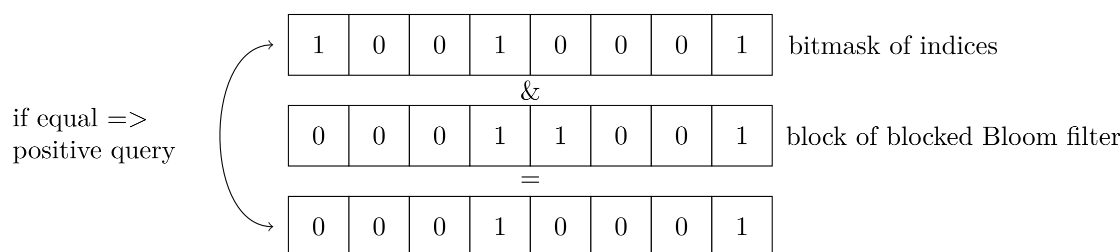


Figure 3.3 Checking of blocked Bloom filter using bitmask

And when inserting an element, insert it only inside one block. The block should be the size of the cache line to guarantee that will fetch the whole block into the cache.[12]

To get the block we want to insert the element into, we hash the element and then take the modulo of the number of blocks. Then we can pretend that the block is a small separate Bloom filter and insert or check the element in the block. The false positive probability of one block can be calculated the same way as in the section 3.1. The problem is that we do not know how many elements are in each block. What we know however is that the load of the blocks is binomially distributed, which we can closely approximate with Poisson distribution. This is the formula:[11]

$$\sum_{i=0}^{\infty} Poisson_{\frac{B}{c}}(i) \times f(B, ik)$$

Where B is the number of bits in the block, c is the number of bits per element $\frac{m}{n}$, k is the number of hash functions and $f(B, ik)$ is the false positive probability of the block with i elements.

3.3.4.1 Register Bloom filter

Another possible improvement to the blocked filter is something called a register Bloom filter. It works by setting block size B to the size of computer register. So for most computers, the block size would be 64 bits. This allows us to use a lot of optimizations. By having a Bloom filter in one register, we can use bitmasks to check/insert multiple bits at once. (3rd bit set to 1 means that the 3rd index in the block should be checked) To leverage this, we want to precompute some bitmasks and store them in a lookup table. Then we use only one hash to get the bitmask from the lookup table for the insertion/query of an element. However, this can increase the false negative probability, because of more collisions in the lookup table. To combat this, we can reserve 6 bits (for a 64-bit register) in the hash that is used for the lookup table. And use these 6 bits to shift the bitmask from the table to achieve greater variety.[12] With this approach, we only need to perform one hash function instead of k.

Compared to the blocked Bloom filter with a block size of the cache line, this method is faster but wastes memory bandwidth because a significant portion of the cache line will not be used. Also, it has a higher false positive probability rate.

In Fig. 3.3 we can see how to check the blocked Bloom filter using bitmasks.

These Bloom filters are only useful for applications with extremely high throughput. For more general use cases, the standard Bloom filter is more space-efficient at the same false positive probability.

Chapter 4

Implementation of Bloom filters in NGINX

Now that we researched Bloom filters, we can finally use this knowledge to design and implement cache summaries in NGINX. First let me introduce NGINX and some of its internals, so we can understand how to extend it.

4.1 NGINX

NGINX is an open-source, high-performance HTTP server and reverse proxy. In this thesis, we use it as a caching reverse proxy. It is widely used because of its performance, stability, and simple configuration. Also, it is highly extensible thanks to its modular architecture, so we can easily add new functionality through 3rd party modules. Furthermore, NGINX comes equipped with built-in modules for most regular use cases.[1]

Our focus will be on the NGINX proxy module. This module is responsible for proxying the request to the specified location. It can be used as a reverse proxy, load balancer, caching proxy, etc. Let us look at the configuration of a minimal caching reverse proxy in NGINX (Listing 4.1).

Directives that start with "`proxy_`" are from the proxy module.[16]

- **proxy_cache_path** - Creates a new shared memory (Subsec. 4.1.1) for cache entries and specifies the directory where the cache files will be stored.
- **proxy_pass** - Specifies where to forward the request. In this case to the backend server.
- **proxy_cache** - Enables caching and specifies what cache key zone to use. The cache zone can also be specified dynamically for each request using variables.

  NGINX uses event-driven, non-blocking architecture, which leads to good and predictable performance. It always has only one worker per process. Ideally, you want to have as many worker processes as you have CPU cores. Because of this architecture, it is crucial that all operations in workers are non-blocking. For example, if we perform some blocking I/O in a worker when processing a request, we will block the whole worker until the I/O operation is finished. No other requests can be processed in the meantime on this worker.

  As each worker is in a different process (so different memory space), we need a way to share data between them.

Code listing 4.1 Minimal caching reverse proxy configuration.

```
proxy_cache_path /path/to/cache keys_zone=cache_zone:10m;
http {
    server {
        listen 80;
        server_name example.com;

        location / {
            proxy_pass http://backend_server;
            proxy_cache my_cache;
        }
    }
}
```

Code listing 4.2 Definition of NGINX struct for shared memory.[17]

```
typedef struct {
    u_char      *addr;
    size_t       size;
    ngx_str_t    name;
    ngx_log_t   *log;
    ngx_uint_t   exists;   /* unsigned  exists:1;  */
} ngx_shm_t;
```

4.1.1 Shared memory

NGINX provides a way to share data between workers using shared memory. It is used in a lot of places in the NGINX source code. For example, `proxy_cache_path` directive creates a shared memory that will hold the content of the cache. This is the shared memory zone that we are most interested in because we need to track the contents of the cache in the Bloom filter. So we need to find all the places in the source code where the cache inserts, updates or deletes the content and add our logic for updating the Bloom filter.

In the source code, shared memory for Linux is represented by this structure Listing 4.2.

- **addr** - Pointer to the start of the actual shared memory. On Linux it is an anonymous mmap[1]
- **size** - Size of the shared memory.
- **name** - Name of the shared memory.
- **log** - Shared memory log.
- **exists** - Flag that indicates if the shared memory exists.

This structure is platform-dependent because processes are implemented differently on different operating systems. For Windows, this struct has some extra fields, but we are not focused on Windows.

[1] More info about info about mmap and MAP_ANONYMOUS option on: https://man7.org/linux/man-pages/man2/mmap.2.html

Code listing 4.3 Definition of file cache struct in shared memory.[17]

```
typedef struct {
    ngx_rbtree_t                      rbtree;
    ngx_rbtree_node_t                 sentinel;
    ngx_queue_t                       queue;
    ngx_atomic_t                      cold;
    ngx_atomic_t                      loading;
    off_t                             size;
    ngx_uint_t                        count;
    ngx_uint_t                        watermark;
} ngx_http_file_cache_sh_t;
```

We also need to know about shared memory zones, because our Bloom filter also needs to be shared between workers for updates and reads. So later, when we decide on the specific Bloom filter, we will create a shared zone for it.

4.1.2 HTTP file cache

Most of the code that is responsible for caching in the proxy module is delegated to the `http_file_cache.c` file. Let us look at the most important structures and functions in this file that we need to know about for our implementation.

In Listing 4.3 is the main structure of the proxy module that is shared between workers. I will explain the most important fields of this structure.

- **rbtree** - Red-black tree that holds the cache entries. On every request, we search the tree for the cache entry.
- **sentinel** - Sentinel node of the red-black tree. When iterating over the tree, we stop when we reach a node that is equal to the sentinel.
- **queue** - LRU Queue of cache entries. Is used for cache eviction of old entries when the cache is full. Every time a cache entry is accessed, it is moved to the front of the queue.
- **cold** - Flag that indicates if all cache entries from disk are loaded into memory.
- loading - Flag that indicates if the cache is currently loading cache entries from disk.
- **size** - Size of the cache in bytes.

Apart from worker processes, there is also a cache loader and cache manager process. The cache loader is run only on startup and is responsible for loading all cache entries from disk to memory. The cache manager is responsible for cache eviction in the background and other maintenance tasks.

So what we are interested in is the *rbtree* and *queue*. Because it holds the cache entries we want to store in the cache summary.

Let us look what does elements of these containers look like. (I have added comments for every field. Information about these fields was acquired by reading NGINX source code[17])

The Listing 4.4 holds the structure that represents a cache entry. On every cache entry we have a reference to the rbtree node and queue node. The rbtree node contains a key (used for searching), parent, left and right child node and color (used for balancing purposes, not important for us). The queue is a doubly linked list, so we can easily remove and insert elements from arbitrary positions. This means that from any given cache entry, we can see the element immediately behind us and the element immediately in front of us in the LRU queue.

The best place to update the Bloom filter would be every time after flag `exists` is updated. This means that when `exists` is set to 1, cache entry is now cached and ready on the disk, so we will add it to the Bloom filter. When `exists` is set to 0, the cache entry is removed from the cache, so we will remove it from the Bloom filter.

4.2 Bloom filter implementation

The First idea as a proof of concept was to use the standard Bloom filter3 and periodically rebuild it. It would not be ideal, but it would suffice as a proof of concept. Squid also uses this approach. But it turned out that in our case, this approach is not feasible. Because only two structures that hold the cache entries (which we need to rebuild the Bloom filter) are the red-black tree and the LRU queue. Both of these structures are not suitable for iterating over all elements. The problem is that they are shared between workers, so we need to lock them for the whole time of the iteration. This could have a huge impact on performance given that there could be millions of cache entries, and we are not iterating over them efficiently in terms of cache locality. (jumping between nodes in the tree)

If we did not lock these structures during the iteration, we could get into a situation where our current node is deleted or moved to another position in the tree. Squid can afford this approach because it stores cache entries in a hash table with linear probing. So it can iterate over all elements without locking the whole structure.

This left us to use counting Bloom filter so we can remove elements from summaries. We created a new file `ngx_bloom_filter.h` that will be included in the core NGINX module. In Listing 4.5 we can see the definition of the structures for Bloom filter.

For the counting Bloom filter, we have chosen to use a byte-sized counter. Not because of potential counter overflow, but for simplicity of first implementation and because of better atomicity of updating the counter. To increment a byte counter, we use atomic increment operation (`__sync_fetch_and_add` on gcc). If we used 4-bit counters, we would have to first fetch the byte, mask out only the 4 bits that we want, increment them and then store it back. This is not atomic, so we would need to either lock the whole byte or use something like compare_and_swap operation. We prioritized performance over memory usage.

Updating Bloom filter can look something like this Listing 4.6. We receive k already computed hash values for the element. Then atomically increment the counters at the computed indices. It is important to check for counter overflow/underflow. Especially in this proof of concept, we can catch bugs like deleting non-existing elements, inserting the same element multiple times, or having a bad hash function.

For our choice of k we decided to set it statically to 4. This is pragmatic choice, because we already have 16bytes of cache entry key. This key is md5 hash of the requested URL and possibly some other things like headers. It is used to identify the cache entry. We will just divide those 16 bytes into 4 parts and modulo them by the size of the Bloom filter. Because of this we do not have use any extra hash functions.

In Listing 4.7 we can see the function that gets the hash values for the Bloom filter.

Code listing 4.4 Definition of NGINX struct representing cache entry.

```
typedef struct {
    ngx_rbtree_node_t  node; // red black tree node
    ngx_queue_t        queue; // queue node

    // part of the cache key - second part is hidden in node->key
    u_char             key[NGX_HTTP_CACHE_KEY_LEN
                           - sizeof(ngx_rbtree_key_t)];

    // refferece counter - do not free this object if its not zero
    unsigned           count:20;
    // number of uses of this cache entry - we can
    //   have rules to cache the file only ifit was used more than X times
    unsigned           uses:10;

    unsigned           valid_msec:10;
    // error code
    unsigned           error:10;
    // if the file is valid and physicaly exists on disk
    unsigned           exists:1;
    // if the file is being updated
    unsigned           updating:1;
     // if the file is being deleted
    unsigned           deleting:1;
    // if the file was purged  - that means someone specifically
    //   requested to remove this file from cache
    unsigned           purged:1;

    /* 10 unused bits */

    // unique file id
    ngx_file_uniq_t    uniq;
    // when this cache entry expires
    time_t             expire;
    // max-age in the cache-control header
    time_t             valid_sec;
    // size of the body of cached request
    size_t             body_start;
    // size of the file on the filesystem
    off_t              fs_size;
    // how long this cache entry is locked - cache entries
    //   are shared betwen workers, so we need to lock them before using
    ngx_msec_t         lock_time;
} ngx_http_file_cache_node_t;
```

Code listing 4.5 Definition of Bloom filter structs.

```
/* statistics for the counting Bloom filter */
typedef struct {
    ngx_atomic_t  nelts;      /* number of elements */
    ngx_atomic_t  bits;       /* number of bits set to 1 */
    ngx_atomic_t  overflows;  /* number of counter overflows */
    ngx_atomic_t  underflows; /* number of counter underflows */
} ngx_bloom_filter_stats_t;


/* main counting Bloom filter structure */
typedef struct {
    ngx_bloom_filter_stats_t   stats;
    ngx_uint_t                 size; /* size of the Bloom filter */

    /* byte array representing counting Bloom filter */
    u_char                     arr[];
} ngx_bloom_filter_t;

/* structure for easier manipulation with Bloom filter keys */
typedef struct {
    ngx_uint_t                *keys;
    size_t                     len;
} ngx_bloom_filter_keys_t
```

Code listing 4.6 Updating the counting Bloom filter.

```
ngx_int_t
ngx_bloom_filter_update(ngx_bloom_filter_t *bf,
    ngx_bloom_filter_keys_t *keys, ngx_int_t add)
{
    ngx_uint_t  i, initial, *key;

    key = keys->keys;

    ngx_atomic_fetch_add(&bf->stats.nelts, add);

    for (i = 0; i < keys->len; i++) {
        initial = ngx_atomic_fetch_add(&bf->arr[key[i] % bf->size], add);

        if (initial + add > 255) {
            ngx_atomic_fetch_add(&bf->stats.overflows, 1);
        } else if ((ngx_int_t) initial + add < 0) {
            ngx_atomic_fetch_add(&bf->stats.underflows, 1);
        }
    }

    return NGX_OK;
}
```

Code listing 4.7 Function to get hash values from cache entry key.

```
void
ngx_http_bloom_filter_bf_fill_keys(ngx_http_file_cache_node_t *fcn,
ngx_bloom_filter_keys_t *keys)
{
    ngx_uint_t  i;
    ngx_uint_t *k;

    memset(keys->keys, 0, sizeof(ngx_uint_t) * 4);

    k = keys->keys;

    // convert 8 byte array into 2 32bit integers
    for (i = 0; i < 4; i++) {
        k[0] |= (ngx_uint_t) fcn->key[i] << (i * 8);
        k[1] |= (ngx_uint_t) fcn->key[i+4] << (i * 8);
    }

    // convert 64-bit integer into 2 32bit integers
    k[2] = (uint32_t) fcn->node.key; // lower 32bits of key
    k[3] = fcn->node.key >> 32; // upper 32bits of key
}
```

We receive the cache entry node we talk about in subsection 4.1.2 and get half of the key from *key* and the second half from *node→key*. If we found out in future that $k = 4$ is not enough, we would have to compute some more hash data from the 16-byte cache key.

4.3 Bloom filter module

Our Bloom filter implementation is done, now we just need to integrate it into NGINX. We will create a new module for NGINX that will be responsible for creating and managing Bloom filters.

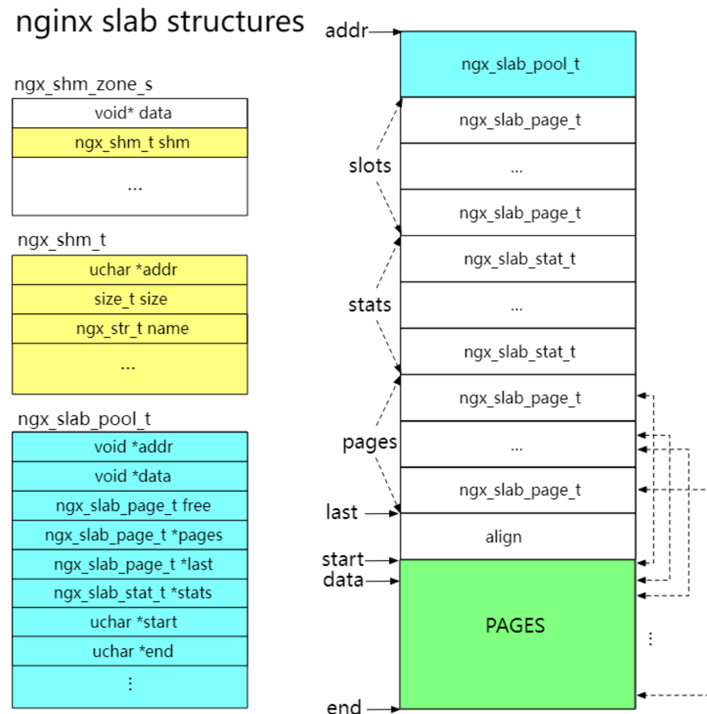
4.3.1 Allocations in shared memory

First, we need to create a shared memory zone for our Bloom filter. Adding the shared zone is simple, we just need to call `ngx_shared_memory_add` function in one of our functions in the module that is called at startup. Creation of shared memory zone is demonstrated in Listing 4.8.

Code listing 4.8 Adding shared memory zone for Bloom filter.[18]

```
ngx_str_t        name;
ngx_foo_ctx_t   *ctx;
ngx_shm_zone_t  *shm_zone;

ngx_str_set(&name, "foo");

/* allocate shared zone context */
ctx = ngx_pcalloc(cf->pool, sizeof(ngx_foo_ctx_t));
if (ctx == NULL) {
    /* error */
}

/* add an entry for 64k shared zone */
shm_zone = ngx_shared_memory_add(cf, &name, 65536, &ngx_foo_module);
if (shm_zone == NULL) {
    /* error */
}
/* shared zone is not allocated yet, it will be done by
   NGINX at the end of configuration setup */

/* register init callback and context */
/* when the callback is called, the shared zone is already allocated, so we can use it */
/* in ctx we will save necessary information for Bloom filter creation */
shm_zone->init = ngx_foo_init_zone;
shm_zone->data = ctx;
```

To allocate from the shared memory we have to use `ngx_slab_alloc` function. This function performs slab allocation on the already mmaped shared memory. Slab allocation is also used in the Linux kernel[19], but this is a simpler version. When we want to allocate a 1 MB Bloom filter from shared memory we call ngx_slab_alloc(shm→shpool, 1MB), but there is a catch. If we created a shared memory zone with size of 1 MB, we cannot allocate 1 MB of our structs from it. This is because there are already some structures for managing the shared memory allocated in it, and also the allocation is done from pages. As we can see in Fig. 4.1 at the beginning of the shared memory, there are these pre-allocated structures. Most notably slots that are responsible for allocating data of size 2^{slot_index} bytes. (Every allocation needs to be power of two) If we want to allocate, for example, 17 bytes, slot $5(2^5 = 32)$ will be used and 32bytes will be allocated. The 15 bytes will be wasted. Each slot manages it is own un-filled pages. Pages can only contain allocations from one slot. (So every allocation in one page has to be the size) If we want to allocate more than half of page, we will allocate whole pages from linked list of free pages directly, without using slots. Given that the page size is 4 KiB (can be changed), to allocate 1 MiB of memory, we would need to allocate 256 pages plus some extra space for the 256 `ngx_slab_page_t` structures.[20]



■ **Figure 4.1** NGINX shared memory overview[20]

To know how much memory we need for structures in our shared memory, we reverse engineered the NGINX source code[17] for allocation and came up with this formula:

```
#define ngx_http_bloom_filter_bf_pages_needed(size) \
    ((size) >> ngx_pagesize_shift) + (((size) % ngx_pagesize) ? 1 : 0)
```

It accounts for half-empty pages. This is the upper estimate of pages needed for the allocation of struct with *size* because if the allocation is from available slots, it possibly will not need any extra pages. Now, to compute the size of the shared memory, we compute the pages needed for our structures and add one more page for the shared memory structures (slots, etc.). Then we multiply the number of pages by the page size and the size of structure for managing the page.

```
shm_pages * (ngx_pagesize + sizeof(ngx_slab_page_t));
```

We have the shared memory allocated and can use it for our Bloom filter.

4.4 New directives

We need to create NGINX directives that would allow us to configure and download the Bloom filter.

For this, we have created two new directives:

Code listing 4.9 Nginx config with Bloom filters.

```
proxy_cache_path /path/to/cache keys_zone=cache_zone:10m bloom_filter_zone=bf;
bloom_filter bf 1m;
http {
    server {
        listen 80;
        server_name example.com;

        location / {
            proxy_pass http://backend_server;
            proxy_cache my_cache;
        }

    }
    server {
        listen 8080;
        server_name bf.example.com;

        # to download Bloom filter use:
        # curl http://bf.example.com:8080/bloom_filters/<bloom_filter_name>
        location = /bloom_filters {
            bloom_filter_download;
        }
    }
}
```

- **bloom_filter** - Specifies the name of shared memory (also the Bloom filter) and size of the Bloom filter.

  Used like this: **Bloom_filter foo 1m;**
- **bloom_filter_download** - Needs to be in a context of some location block. Will return the standard Bloom filter computed from our counting Bloom filter.

Also, we need to add an option to the `proxy_cache_path` directive to specify what Bloom filter to use. In Listing 4.9 is the minimal configuration to set up Bloom filters.

Because downloading counting Bloom filter would waste a lot of bandwidth, we will periodically also build the standard Bloom filter and make it available for download. We do not need to know about counters on other servers, because we are downloading only a read-only version of the filter. This is quite fast and efficient, because we only iterate over the counting filter and set the bits in the corresponding places in the standard filter. We are iterating sequentially over the counting Bloom filter so we have good cache locality. It would also be possible to compress the filter, but for simplicity of implementation we will just build the standard filter. Compressing the standard filter would not be very efficient, because if the filter is optimally filled it would not provide much compression.[21] Optimally filled means that approximately 50 % of bits are set to 1.[2] By using hash function the bits should be set sufficiently randomly. This would be like compressing random noise, which is not very compressible.

[2] By optimally filled we mean that we chose optimal k for a combination of m and n.

Chapter 5

Design and implementation of summary sharing

The Bloom filter is now implemented and integrated into the NGINX server. What we need to do is to periodically download the Bloom filter from specified neighbors and use these filters when choosing the upstream server. Doing something like this could be hard if we did it directly in the C NGINX module. But fortunately, we use a lua-nginx-module that allows us to write Lua scripts that can be executed in the NGINX worker. We already have a codebase of Lua scripts that is responsible for dynamic selection of upstreams and more high-level logic. So we will be integrating the Bloom filter sharing logic into these scripts.

5.1 lua-nginx-module

The lua-nginx-module is being developed by a distribution of NGINX called OpenResty. The motivation for this module was to transform NGINX into a fully-fledged web application server for more flexibility. Lua was chosen as the scripting language because it is simple, lightweight, allows compilation at runtime, and most importantly, is easily embeddable into existing applications. Lua code is also more suitable for writing high-level logic than C code. It has dynamic typing, automatic memory management and higher-level abstractions.

The module takes advantage of NGINX event-driven, non-blocking architecture, and runs code in worker processes. This also means that we should not use any blocking I/O operations as it would block the whole worker process. We can also take advantage of existing NGINX modules and just extend them with Lua scripts. The running of Lua code is performed by a custom enhanced version of LuaJIT, which is a Just-In-Time compiler for Lua. Just-In-Time compiler means that the Lua code is compiled to machine code at runtime. It also has a lot of optimizations based on the code that is executed.[22]

5.1.1 Directives

Lua NGINX module provides a lot of directives for running Lua code. They differ in when the code is executed and in what context.

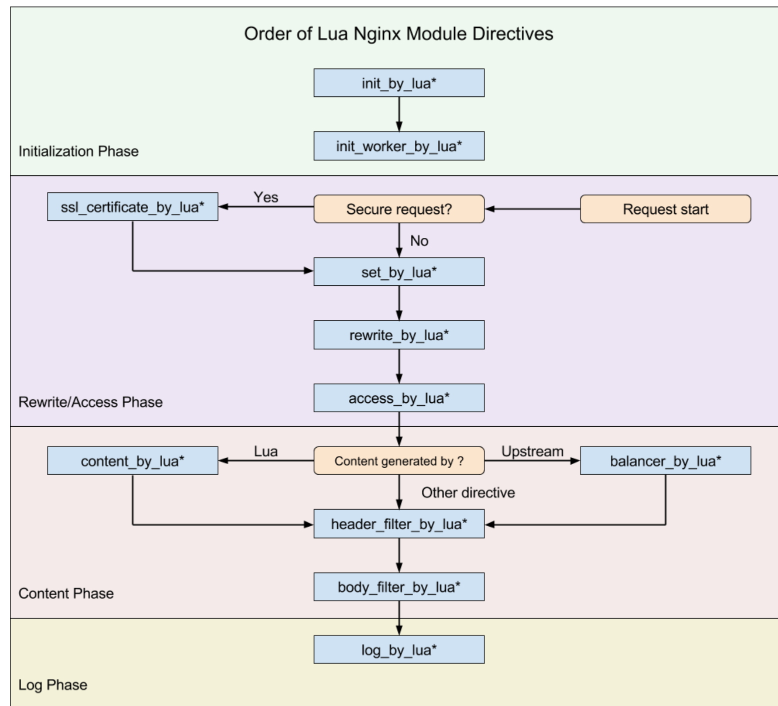


Figure 5.1 OpenResty directives and their phases.[23]

As we can see in Fig. 5.1, the code can be executed in these phases:[23]

- **Initialization phase** - This phase is executed only once when the NGINX worker is started.

  init_by_lua is executed in the master process on startup and `init_worker_by_lua` is executed in every worker process on startup.
- **Rewrite/Access phase** - This phase is executed for every request before the request is fully processed and proxied to upstream. It is executed only by the worker who is processing the request.

  for example `rewrite_by_lua` could be used for URL rewriting or validation. In case of an invalid URL, it could return 404.
- **Content phase** - This phase is executed when it is time to generate and return the response to the client.

  Most interesting for us is the `balancer_by_lua` directive that is executed when we need to dynamically choose the upstream server of the proxy. There could be some complex logic or even dynamic loading of upstream servers. We will use this directive to prioritize upstreams based on the Bloom filter.
- **Log phase** - This phase is executed after the response is sent to the client.

  We could use this phase to log some information about the request.

5.1.2 FFI

Lot of libraries and functions in our Lua scripts are implemented in C and are called through foreign function interface (FFI). FFI is a mechanism that allows us to call functions from one language in another language. This is one of the main reasons why Lua was

chosen, because it is very easy to call C functions from Lua. Apart from using existing libraries, we can also easily write our own C code and call it from Lua.[24]

5.1.3 Worker synchronization

Because we are running Lua code in different worker processes, we face the same problem as in C code. The problem is that we also cannot directly access shared memory structures defined in C code. Only if we defined our own FFI functions for working with the shared memory.

To solve this problem, there is already a library for this. Lua NGINX module provides a shared dictionary that allows us to store and retrieve data between workers. This shared dictionary is implemented in C, and in Lua, we just call it through FFI calls. Everything done in C code to the shared dictionary is done under mutex, so this library is thread-safe.

It can also be used for synchronization between workers. Unfortunately, it was not made for this because there is nothing like a mutex or semaphore. For synchronization, we have to use the `add` function that saves a value to the dictionary only if it is not already there, `delete` and `increment` function. We cannot do a classic lock, but we can, for example, block other workers from doing something by setting some key in the dictionary using the add function. Because only one of the workers will succeed in setting the key, the others will fail.

While implementing the Bloom filter sharing mechanism, we encountered some complex synchronization issues that could not be easily resolved with the available functions. Consequently, we had to expand the shared dictionary with compare-and-swap functionality. Listing 5.1.

5.2 Bloom filter sharing

What we need to do is to periodically download the Bloom filter from specified neighbors and use these Bloom filters to choose the right upstream server.

5.2.1 Downloading the Bloom filters

For periodic tasks, there is already ngx.timer.every function in lua-nginx-module. This functions allows us to execute custom function every n milliseconds. However this function has some caveats. Mainly that it cannot be used in every context (directive). We cannot run this function in `init_by_lua` which runs only in the master process. What we have to do is run this function in `init_worker_by_lua`. This is not a big problem, we will have a timer in every worker process and will have to do some synchronization between workers. We can also leverage this and download the Bloom filter in parallel, in every worker process.

First problem that we have to solve is where to store the downloaded Bloom filters. Share dictionary may seem like a good choice, but unfortunately we cannot use it. Because Lua shared dictionary does not return strings by reference, but by value. We could have very large Bloom filters and allocating every request whole new string would be very inefficient.

We have decided to create a new shared memory zone for them in our C Bloom filter module. This shared memory zone has a fixed amount of slots and every slot could hold one Bloom filter. The management of which slot is used by which Bloom filter is done in the Lua shared dictionary.

One more problem with the downloading we had to solve was how to store the downloaded Bloom filter. We cannot perform the downloading only from Lua, because it would buffer the whole response (potentially more than 100 MB) in memory. Solution to this problem is to create another location block with `proxy_pass` and our custom body filter (function for processing streaming data) directive that we will implement in our Bloom filter module. We will then request this endpoint from Lua and the body filter will download the response with chunked encoding. As the chunks are read, we immediately write them to the shared memory and then free the already read chunks.

The body filter is implemented in C and registered into the chain of already existing NGINX body filters. Every NGINX module that performs some operation on the response body must have registered a body filter. The body filter is called for every chunk of the response body, it allows us to read/modify the response body in a streaming fashion.

The location would look something like this listing 5.2.

The tokens that start with "`$arg_`" are variables that are evaluated every request. They will be replaced by query arguments in the request. In Lua to download the Bloom filter into shared memory in slot 0, we would call this Listing 5.3.

Code listing 5.1 Compare-and-swap function for shared dictionary.

```
--- @return success bolean, error string,
---         boolean that indicates if we had to expire
---         some keys to fit them in the shared dictionary
local function shdict_compare_and_swap(zone, key, new_value, exptime,
    flags, old_value, old_flags)
    -- checks if the zone exists and is valud, else throws error
    zone = check_zone(zone)

    if not old_value then
        return nil, "nil old_value"
    end
    if not old_flags then
        old_flags = 0
    end
    if not flags then
        flags = 0
    end
    -- stop garbage collection,
    -- so we can safely lock the shared dictionary

    -- if we did not stop it, the garbage collector could potentially
    -- try to access the shared memory and lock it again, which would
    -- result in a deadlock
    collectgarbage("stop")
    shdict_lock(zone) -- we modified also the c code
                      -- to allow us to lock the shared dictionary directly

    -- we need to call these functions with pcall, because they could
    -- throw error, and that would leave the shared dictionary locked
    local ok, val_or_err, flags = pcall(shdict_get_locked,zone, key)

    if ok and val_or_err == old_value and flags == old_flags then
        local ok, succ_or_err, err, forced = pcall(shdict_set_lockedzone,
                                              key, new_value, exptime, flags)

        shdict_unlock(zone)
        if not ok then
            error(succ_or_err)
        end

        return succ, err, forced
    end

    shdict_unlock(zone)

    if not ok then
        error(val_or_err)
    end

    return false, "not matched", false
end
```

Code listing 5.2 Location block for downloading Bloom filter into shared memory.

```
location = /download_bloom_filter {
    bloom_filter_body_filter on;
    proxy_set_header Host $arg_bf_url:$arg_bf_port;
    proxy_cache off;
    proxy_pass http://$arg_bf_url:$arg_bf_port/bloom/$arg_bf_zone;
}
```

Code listing 5.3 Downloading the Bloom filter from Lua.

```
1 local res, err = ngx.location.capture("/download_bloom_filter", {
2                      args = {
3                          bf_url = upstream.addr,
4                          bf_port = port,
5                          bf_zone = upstream.conf.bf_zone or "b",
6                          index = 0
7                      },
8                  })
```

We can now download the Bloom filters, now all that is left is to download them periodically. We set up a timer in `init_worker_by_lua` that will trigger Bloom filter checking every x seconds. In shared dictionary we store the time it will expire and new Bloom filter should be downloaded. If our checking function finds out that the Bloom filter is expired, it will trigger the download of Bloom filter into a free slot in shared memory. And after successfully download and no reader of the old Bloom filter, we will free the old Bloom filter slot. It is important to have at least twice as many slots as there are neighbors, so we can download the Bloom filter from one neighbor while the other is still using the old one.

5.2.1.1 Frequency of downloading

What has to be considered is how often we should download the Bloom filters from neighbors. Choosing too short of a time could lead to unnecessary load on the servers and network. Choosing too long of a time could lead to poor performance of the Bloom filter. The Bloom filter data structure does not have false negatives, but using an older Bloom filter that does not contain more recent data could lead to false negatives. The more files are changed in the neighbor cache without updating our Bloom filter, the more false negatives we will have. But because every server has a different rate of cache updates, we cannot just set a fixed time of downloading. Ideally we would download the Bloom filter every time the neighbor cache summary changed by a certain percentage. For simplicity we chose to set the time of downloading manually based on the expected rate of cache updates. In future we could implement some more advanced logic for downloading the bloom filters or even send only diffs of the filters.

5.3 Selecting the upstream server

We finally have everything we need to prioritize the upstream server based on the Bloom filter. Now we just need to slightly modify our upstream selection logic in `balancer_by_lua` directive.

We have a Bloom filter for every neighboring server, but some servers are just sharded cache like in section 1.2. For these sharded servers we test only the Bloom filter of server that we get from consistent hashing.

So the logic of upstream selection is like this:

1. If we have requested file in cache, return it from cache
2. If we do not have any neighbors, forward to default upstream server
3. If we have neighbors, find first 2 neighbors that contain the requested file in their Bloom filter. Do not check the Bloom filter of all sharded servers. Then try to forward the request to upstreams we got, if first upstream fails, try the second one.

It is also important to prevent possible loop in neighbors if they are connected in a circle. It is possible that no server has the requested file, but due to false positives, they will all think the other server has the file. To prevent this we disallow the server to forward the request again if it was already forwarded to it. We signal that the request was forwarded from the neighbor cache with a simple header in the request.

The checking of Bloom filter is done through FFI functions in our C module.

Chapter 6

Testing and evaluation

Testing is a crucial part of any project as it allows us to iterate faster and catch bugs early on. It is best to write tests early in the development process and run them as often as possible. The earlier we catch a bug, the easier it is to fix it.

Unit tests[1] should be the main building block of most software projects.[25] Unfortunately, NGINX does not have any unit tests. NGINX uses it is own testing framework called `test::NGINX`. Lua module uses a version with more features called `test::NGINX::Socket`[26]. With these frameworks, you specify the configuration of NGINX, the requests you want to send, and the expected responses or expected log outputs. The test framework then starts NGINX with the specified configuration, sends the request, and checks if the response is as expected. There is a lot of Domain Specific Language (DSL) for writing these tests, but you can also write Perl code directly.

Generally, testing of distributed systems can be quite complex. We have to test not only the individual components but also how they work together. It is important to first test the individual components and then test the integration of these components.[27] In our case we can split the testing into two parts:

* **Testing of Bloom filter generation -** We perform requests that will be cached and check if the Bloom filter is updated correctly.
* **Testing of summary sharing -** We will create beforehand some Bloom filters and then test if the corrected upstream server is chosen.

All of these tests can be run both locally and in the continuous integration pipeline.

Due to the complex nature of the system, it is also important to test the whole system in an environment similar to production. We set up a testing environment consisting of a few servers and tried to simulate some real-world traffic.

In this environment, we can manually monitor more complex things like CPU, memory, or network usage. To name an example, one bug was caught thanks to an in-depth analysis of the system was an incorrect calculation of Bloom filter indices. After downloading the Bloom filter from one of the servers and performing a statistical analysis of it. Our analysis revealed that the filter was not as random as we had expected. Figure 6.1 clearly shows vertical lines in the Bloom filter that have been reshaped into a 2D array.

[1] Unit tests focus on testing only small piece of code in isolated enviroment.

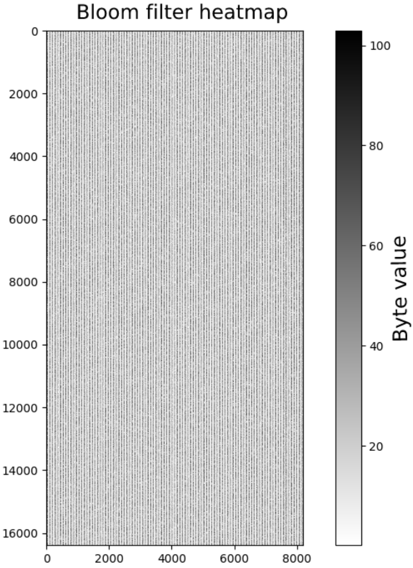


Figure 6.1 Regular pattern in our Bloom filter that indicates something is wrong.

After a deep inspection of the code, we found out that we were miscalculating the indices of the Bloom filter, but it was not caught by our tests because it caused only increased false positives. In Figure 6.2 we can see how the Bloom filter looks after fixing the bug.

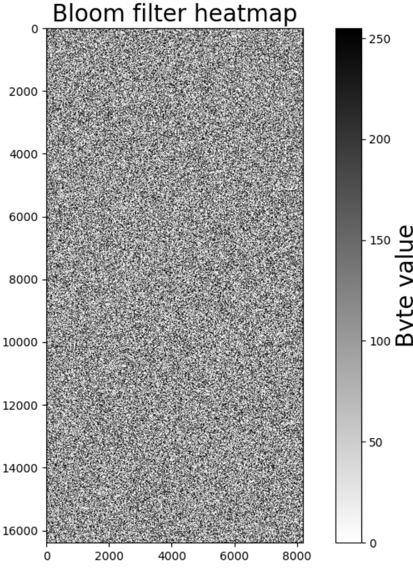


Figure 6.2 Fixed Bloom filter without any irregularities.

6.1 What went wrong

The development of the solution was filled with many challenges. A lot of time was spent on debugging and sometimes even completely rewriting the solution after hitting a dead end. In forthcoming subsections, we will describe some of the problems we encountered.

6.1.1 Downloading of Bloom filter

As mentioned in the subsection 5.2.1, we have a periodic timer that downloads the Bloom filter from neighbors into our shared memory. But this problem was not as straightforward as it seemed. If we just performed a normal HTTP request in the timer to our downloading location (listing 5.2) there could be a problem. When NGINX performs a zero downtime binary upgrade, it starts a new NGINX binary next to the old one and then one by one stops the old workers and starts new workers. In this process, it is possible that a request to download the Bloom filter from the old worker could reach the new worker, thus possibly corrupting existing Bloom filters, because we could download it into the already taken index. NGINX has a function to potentially solve this problem, ngx.location.capture. Instead of performing a normal HTTP request, it performs a subrequest. That is a lightweight request that is processed only in NGINX and does not go through the network. However it has one limitation, it cannot be used in certain contexts. It can be used only in the context of request processing. So it is not possible to use it in the timer function.

To solve this problem we moved the timer function into an endpoint. Now this function is called on every request to the endpoint. The periodicity of the downloading is solved by creating another timer function that requests this endpoint every x seconds. Now if the request to download the Bloom filter reaches the new worker, it will not corrupt the existing Bloom filter because the request does not contain the index of the Bloom filter.

6.1.2 Shared dictionary

In the implementation of the management of Bloom filters, we had to use something to share data between workers. 5.1.3 The shared dictionary is not very good for synchronization between workers as it does not have any locking mechanism. We had to at least implement the compare-and-swap function, but it is not very efficient because it uses locks under the hood. Normally this is an atomic instruction without locks.

There are a lot of possible future improvements to the problem of synchronization between workers and even the shared dictionary itself. We will look into this more in the future because it could be helpful in the other projects too and even possibly to the general public.

6.2 Evaluation

To perform the final test of the solution, we tried to deploy it in the production environment of a CDN provider. For testing purposes, we added Bloom filter generation on few last layer servers in the multilayer cache setup. Also, on one server we added the Bloom filter exchanging logic, so it downloads the filters from other servers. This way, if something went wrong, load balancers could sent the traffic to the other servers.

Bloom filter size was statically set to 128 MB, because each server could have up to 100 millions of cache entries. This gave us worst case false positive probability of 1 %. We deployed the solution for 5 hours and compared the monitored metrics with the previous

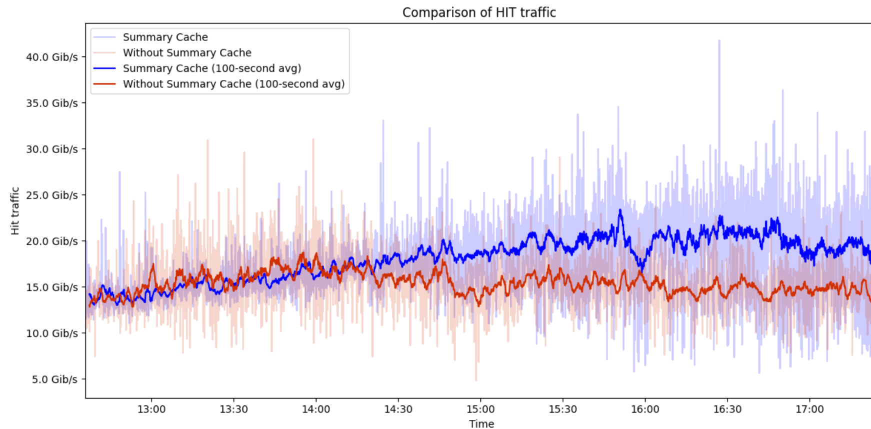


Figure 6.3 Traffic that was served from cache on current server or from neighboring cache.

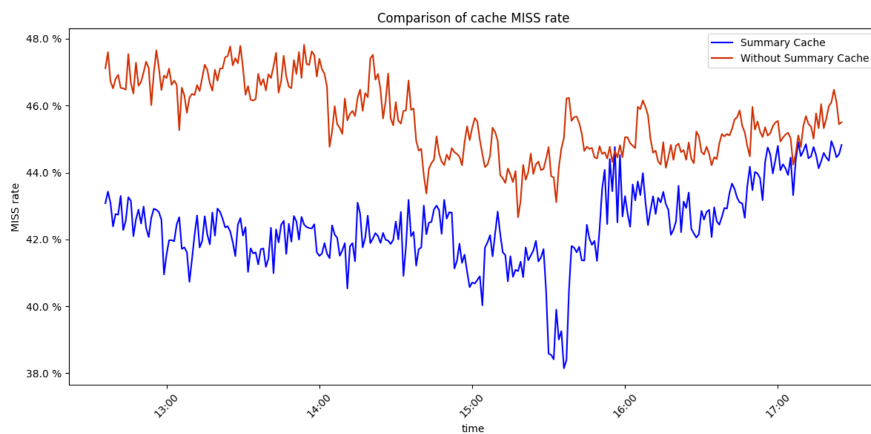


Figure 6.4 Percentage of requests, that were not served from any cache.

day. Metrics that interested us the most were memory usage, cache HIT/MISS ratio and inter-layer traffic.

In Figure 6.3 we can see graph that compares the HIT traffic of our solution with the previous setup. HIT traffic is the data that was served from cache on the current server or from neighboring cache. We can see that for first third of the test the traffic was more or less the same to previous day and then our solution started to outperform the previous setup. These are absolute values, so it could be due to the increased traffic on the network.

More useful metric to evaluate is the cache HIT/MISS ratio as it accounts for the fluctuation of the traffic and tells us how efficient our cache is. In Figure 6.4 we can see that our solution outperformed the previous setup by a few percents. That means, that the close neighboring servers with Bloom filter had only few percents of files that were not in the cache of our testing server. With this information, we can estimate how much origin server traffic would be reduced if we deployed it on all the servers that had Bloom filter. In this case, the origin server traffic would be probably reduced by 2–5 %. It is possible that in different setups of the cache, the improvement would be significantly higher. Also, we did not use all of the servers in the last layer, so this could potentially improve the cache HIT rate even more.

Memory usage on the servers with with bloom filters was negligible compared to the total

memory of the server. Given that the binary Bloom filter is 128 MiB and the counting Bloom filter is 1 GiB. Total additional memory usage is around 1,125 GiB. As the servers have hundreds of GB of memory, this is not a problem.

The inter-layer traffic on last layer increased first of all by the Bloom filter exchange. In this case, we had 4 servers with Bloom filters and one server that was downloading the Bloom filters. The exchange of the filters happen every 60 seconds, so the traffic is increased by $4 * 128\ MiB$ per minute (9 MiB/s). If we deployed the downloading on all of the 5 servers, the traffic would be increased to $(4 * 128\ MiB) * 5$ per minute (45 MiB/s). This is also negligible compared to the multi-gigabit traffic that the servers have. Also, it was increased by the rerouted traffic to the servers with Bloom filters. In this case about 2–5 % of the traffic was rerouted from our single server to the servers with Bloom filters. This would be multiplied by the number of servers that would be downloading the Bloom filters.

Chapter 7

Conclusion

This thesis aimed to improve the cache efficiency of NGINX when used in multilayer deployment.

In the beginning, we got familiar with the multilayer HTTP cache setup. We explained some basic concepts of the setup like what is a reverse caching proxy or consistent hashing. And then we proposed a way to further improve the cache efficiency of this setup. We explored the existing solutions to this problem and decided that none of them is suitable for our use case.

Next, we conducted an in-depth study of Bloom filters, exploring their various applications and tuning for optimal performance. This allowed us to understand what will be necessary for the implementation of our solution.

In the next chapter, we focused on the research of NGINX and its internals. With this knowledge we were able to to implement Bloom filter generation into NGINX.

After that, in the chapter 5, we designed and implemented the Bloom filter sharing mechanism. To do this, we had to explore the options of the Lua NGINX module and implement the sharing logic in Lua.

The final chapter was devoted to the testing and evaluation of the solution that was implemented. We talked about what went wrong, how we fixed it, and what could be improved in the future. We managed to improve the cache efficiency by about 2–5 % in the production environment of a CDN provider.

To sum everything up, we have successfully fulfilled the goal of this thesis by explaining the multilayer HTTP cache setup, studying Bloom filters, and implementing cache sharing mechanism into NGINX. This mechanism also supports cache eviction and purging.

This is the first version of the Bloom filter module, so there is still room for improvement, we made some shortcuts to make the implementation faster. In the future, we could make the module more configurable, research alternatives to Bloom filters, or implement better exchange strategies. Also, we could improve the Lua shared dictionary from which could other projects benefit too.

Bibliography

1. SYSOEV, Igor. *Nginx* [comp. software]. Nginx, Inc., 2023-02. Version 1.25.5 [visited on 2024]. Available from: http://nginx.org/.
2. FIELDING, R; NOTTINGHAM, M; RESCHKE, J. *RFC 9111: HTTP Caching*. RFC Editor, 2022.
3. *HTTP Load Balancer* [online]. F5, Inc., 2024-02 [visited on 2024]. Available from: https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/.
4. KARGER, David; LEHMAN, Eric; LEIGHTON, Tom; PANIGRAHY, Rina; LEVINE, Matthew; LEWIN, Daniel. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In: *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*. El Paso, Texas, USA: Association for Computing Machinery, 1997, pp. 654–663. STOC '97. ISBN 0897918886. Available from DOI: 10.1145/258533.258660.
5. GARRETT, Owen. *Sharding the cache on Nginx Plus web cache servers* [online]. F5, Inc., 2023-01 [visited on 2024]. Available from: https://www.nginx.com/blog/shared-caches-nginx-plus-cache-clusters-part-1/.
6. XIE, Wei; CHEN, Yong. Elastic Consistent Hashing for Distributed Storage Systems. In: *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2017, pp. 876–885. Available from DOI: 10.1109/IPDPS.2017.88.
7. FAN, Li; CAO, Pei; ALMEIDA, J.; BRODER, A.Z. Summary cache: a scalable wide-area Web cache sharing protocol. *IEEE/ACM Transactions on Networking*. 2000, vol. 8, no. 3, pp. 281–293. Available from DOI: 10.1109/90.851975.
8. ROUSSKOV, Alex; WESSELS, Duane. Cache digests. *Computer Networks and ISDN Systems*. 1998, vol. 30, no. 22-23, pp. 2155–2168.
9. WESSELS, Duane. *Squid* [comp. software]. Squid Cache, 1996-12. Version 5.9 [visited on 2024]. Available from: http://www.squid-cache.org/.
10. BONOMI, Flavio; MITZENMACHER, Michael; PANIGRAHY, Rina; SINGH, Sushil; VARGHESE, George. An improved construction for counting bloom filters. In: *Algorithms–ESA 2006: 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006. Proceedings 14*. Springer, 2006, pp. 684–695.
11. PUTZE, Felix; SANDERS, Peter; SINGLER, Johannes. Cache-, hash-, and space-efficient bloom filters. *Journal of Experimental Algorithmics (JEA)*. 2010, vol. 14, pp. 4–4.

12. KRASSOVSKY, Sasha. *Modern Bloom Filters: 22x Faster!* [Online]. 2023. [visited on 2024]. Available from: https://save-buffer.github.io/bloom_filter.html.

13. ABDENNEBI, Anes; KAYA, Kamer. A bloom filter survey: Variants for different domain applications. *arXiv preprint arXiv:2106.12189*. 2021.

14. SHTUL, Ariel; BAQUERO, Carlos; ALMEIDA, Paulo Sérgio. Age-partitioned bloom filters. *arXiv preprint arXiv:2001.03147*. 2020.

15. ALMEIDA, Paulo Sérgio; BAQUERO, Carlos; PREGUIÇA, Nuno; HUTCHISON, David. Scalable bloom filters. *Information Processing Letters*. 2007, vol. 101, no. 6, pp. 255–261.

16. *Proxy module* [online]. 2024-04. [visited on 2024]. Available from: http://nginx.org/en/docs/http/ngx_http_proxy_module.html.

17. *Nginx source code* [online]. 2024-04. [visited on 2024]. Available from: http://nginx.org/download/nginx-1.25.5.tar.gz.

18. *NGINX development guide* [online]. [visited on 2024]. Available from: http://nginx.org/en/docs/dev/development_guide.html.

19. GORMAN, Mel. *Understanding the Linux virtual memory manager*. Vol. 352. 2004.

20. CATBRO666. *Shared memory management in Nginx - the slab algorithm* [online]. 2022-03. [visited on 2024]. Available from: https://catbro666.github.io/posts/2dc32e47/.

21. MITZENMACHER, Michael. Compressed bloom filters. In: *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*. 2001, pp. 144–150.

22. *lua-nginx-module* [online]. [visited on 2024]. Available from: https://github.com/openresty/lua-nginx-module.

23. *Directives - OpenResty Reference* [online]. OpenResty [visited on 2024]. Available from: https://openresty-reference.readthedocs.io/en/latest/Directives/.

24. *lua-resty-core* [online]. 2024-04. [visited on 2024]. Available from: https://github.com/openresty/lua-resty-core.

25. VOCKE, Ham. *The Practical Test pyramid*. Available also from: https://martinfowler.com/articles/practical-test-pyramid.html.

26. *Test::Nginx* [online]. [N.d.]. [visited on 2024]. Available from: https://metacpan.org/pod/Test::Nginx.

27. ULRICH, Andreas; KÖNIG, Hartmut. Architectures for testing distributed systems. *Testing of Communicating Systems: Methods and Applications*. 1999, pp. 93–108.

The contents of the included media

- `thesis.pdf` .... compiled thesis into PDF
- `src` .... Folder with LaTeXsource code