System Design Interview Textbook (2024)

Topic: Distributed Caching

Caching for System Design Interviews

An Illustrated Textbook

✅ Created + Reviewed by FAANG Engineers
✅ Step-by-step Detailed Guide
✅ 102 diagrams
✅ 101 pages
✅ Completely Free 🤑 (yet high quality)

By Harsh Goel
https://www.linkedin.com/in/harsh-fyi/

I write free, high-quality books on System Design.

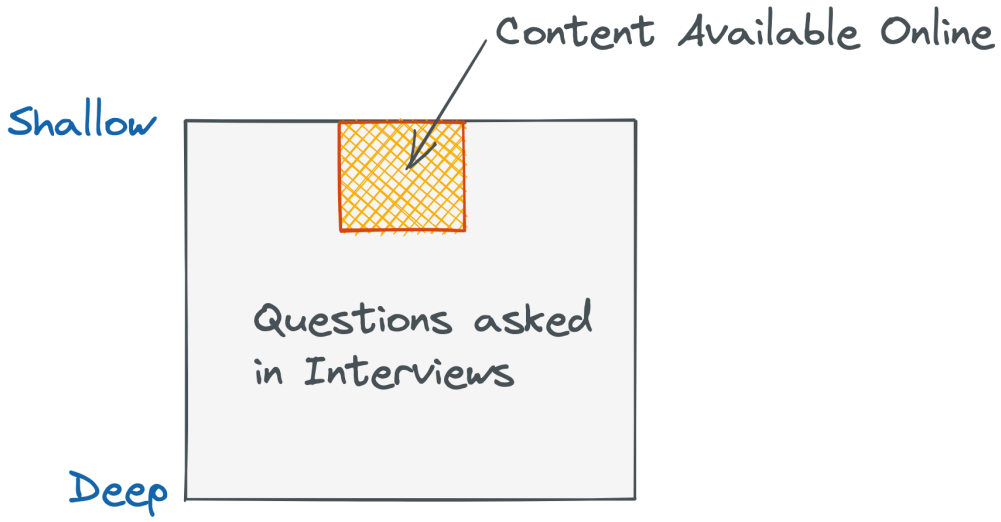
This book is work in progress.
I'm adding sections every week. Get an email when I write a new section:
harsh.fyi/subscribe

🔗 Direct link to this book 📧 Email me here for any questions or suggestions

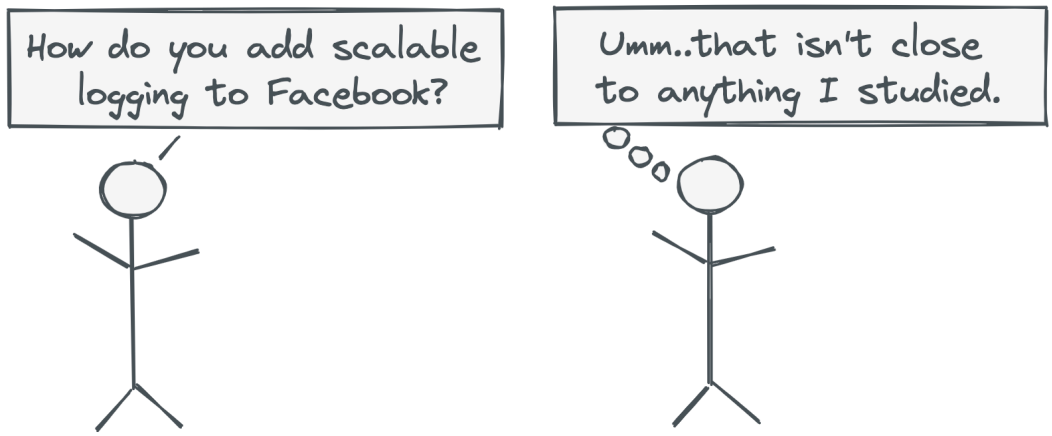
Version 1.0

Why are we writing this book?

After talking to hundreds of candidates at FAANG companies, we found that actual interviews are very different from online prep content.



Companies are going deeper these days. If they ask high level questions like "Design Facebook", they're no longer happy with your standard high-level diagrams you see online. They want to go deeper.



In this book, we try to cover deeper topics that are more relevant to interviews today.

Table of Contents

Ch 1. Caching 101: You run Facebook with 4 machines. Can you make it faster?

Don't skip this chapter!

This chapter's goal is to get you intimate with caches. You probably already know about caches. In this chapter, we get hands on with it.

A big problem with system design is that we read theoretical stuff and never understand it deeply.

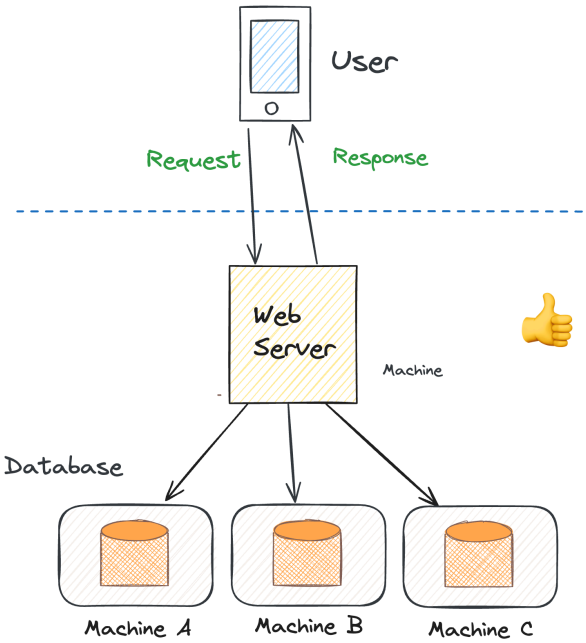
Of course, the best way to learn would be to build a cache yourself, but hey, who has the time? The next best thing is to visualize building it from scratch, which is what we do here.

So yes, don't skip this chapter! It will be helpful for your mindset.

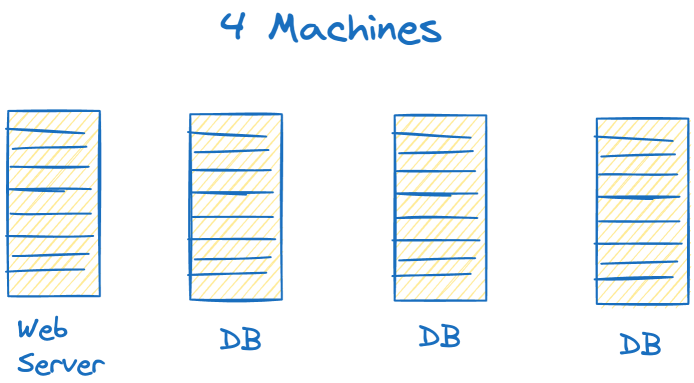
Our 4 Machine Setup at Home

Ok, let's get started. Let's say you run a cute social network like Facebook in 2004. You're a Zuck in the making.

Now let's say that we have a million users - congrats! That's a lot of people. Enough to warrant the following architecture:



You're running this at home, and you have 4 machines: 1 holds your web server and 3 hold your database.

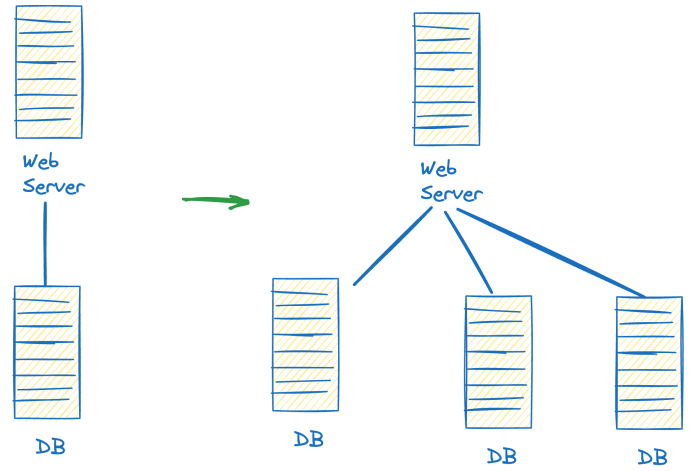


Why do you need 3 machines for your database? Well, it looks like you ran out of space. One machine wasn't enough to hold the massive amount of data your users are generating. What a nice problem to have! VCs are throwing money at you—congrats.

For our purposes, let's say these are commodity desktop CPUs—your everyday CPU from 2004. To make things even more interesting, let's say you bought used machines to save money—they're from 2000.

You turned your desktop machine into a web server in your basement. You took another desktop machine and made it your database. You didn't know the site would grow so fast. As traffic grew, you realized your database was running out of memory. With a 5GB hard drive, users were generating so much data that you ran out of space, so you added 2 more machines to expand your database.

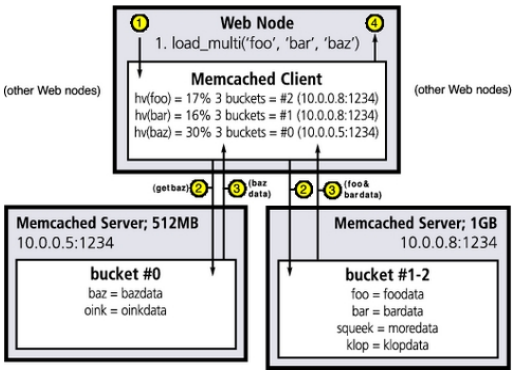
If we're using 3 machines for the database, why just one for the web server? Let's assume that your one web server machine was good enough for the current volume of requests.



So now you have 4 machines—old desktop CPUs—running in your basement, hosting your world-famous social network! What a scenario!

Sounds crazy? It's actually not. Memcached, one of the most popular caching systems today, was created in a similar way.

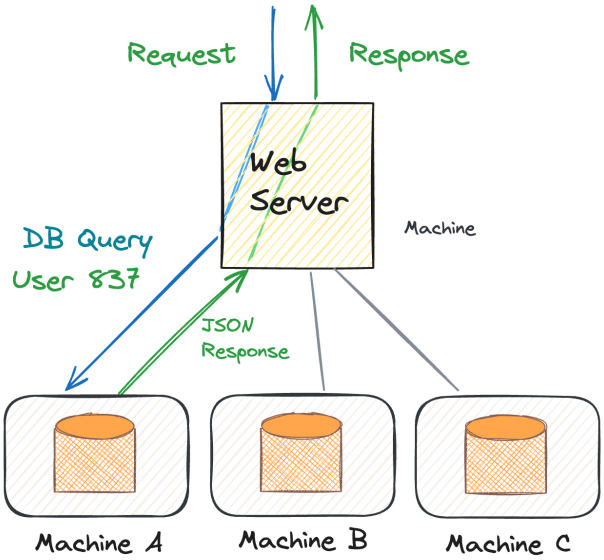
Brad Fitzpatrick created LiveJournal and wanted to speed his site up. So he created a shared cache using commodity machines. At that time, there was nothing like it out there. He described it in his now-famous 2004 article in Linux Journal: [Read the article here]



- Original Memcached Architecture (source)

Ok, let's get back to your site. Things are working well for users. Your architecture is holding up. Requests come to the web server, and they get processed efficiently.

When a request comes to the web server, it usually needs to query the database. This is a common pattern. If you didn't know that, you must be a true beginner to computer science, and I'm glad you're reading this book!



For example, if the request queries user ID 837's profile, you might query the UserProfile table from the MySQL database with something like:

```
SELECT * FROM UserProfile WHERE userID = 837;
```

The web server sends the query to the database machine that holds the data. The machine processes the query and returns the results to the web server.

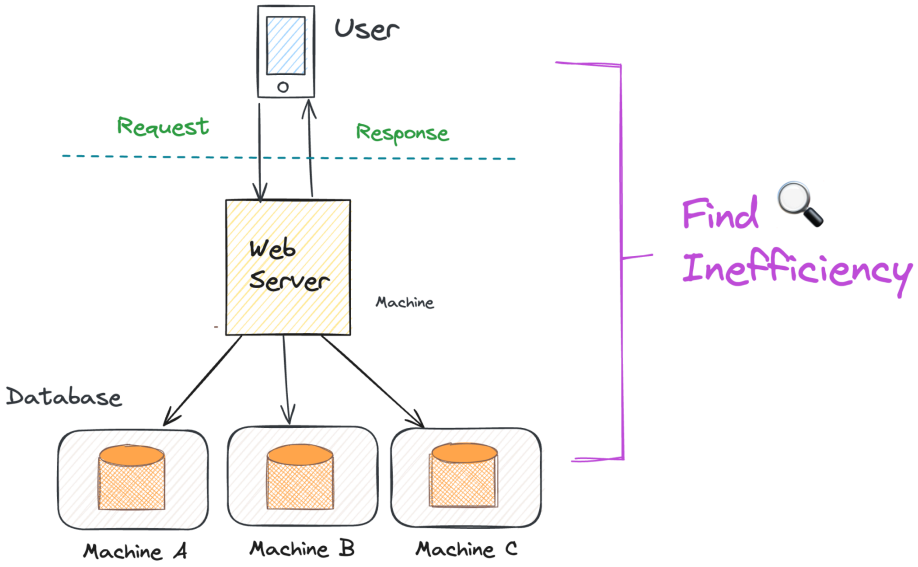
How do we find which of the 3 machines has the data? We'll leave that out for now. You'll get an idea in later chapters. For now, just know that the web server successfully retrieves and processes the data needed to respond to user request.

Things are looking good so far. The site is growing. But now, I ask you, where can you see inefficiencies in the system? Where do you see requests being slow and taking time? That's the whole point of this book. Let's get into it.

Why is this system slow? Studying Inefficiencies

Don't skip this section!

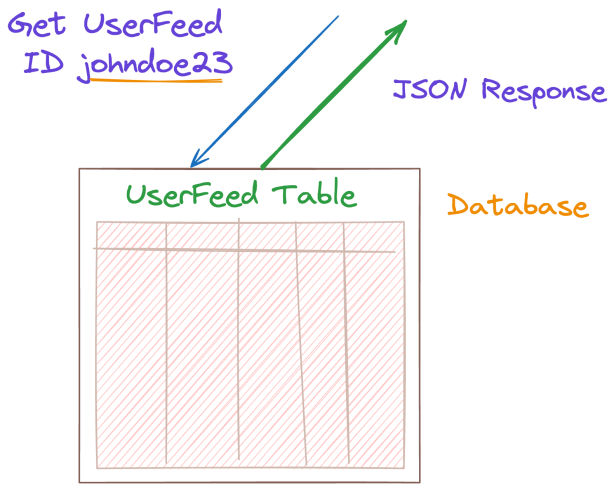
This is a good exercise to observe and learn. In most interviews, you'll be expected to critique your own system - find bottlenecks, performance issues, etc.



Let's trace a request.

It comes in and hits the web server. The first issue is that this web server is in your basement, so if the user is on another continent, it will take time to get there. That's one inefficiency, but we'll ignore that for now.

Let's say the request is to fetch a user's home feed, like facebook.com/johndoe23/feed. The web server needs to query the Database's UserFeed table.

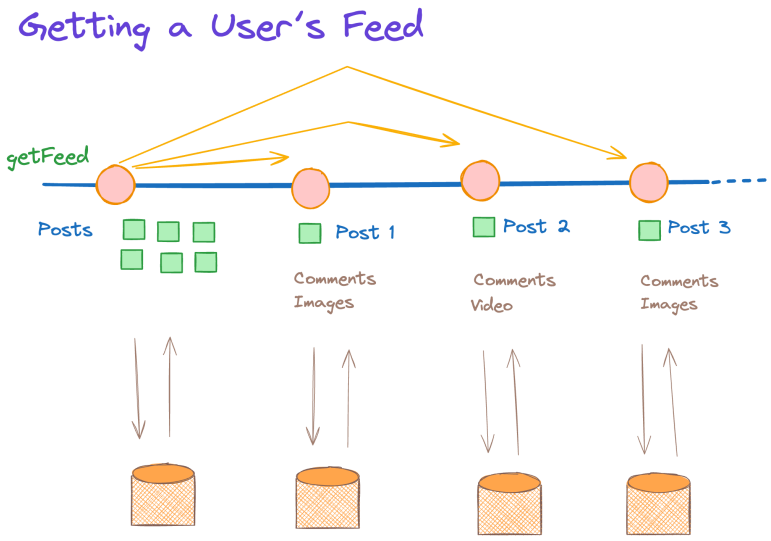


The UserFeed table returns a JSON, which contains post IDs of the posts.

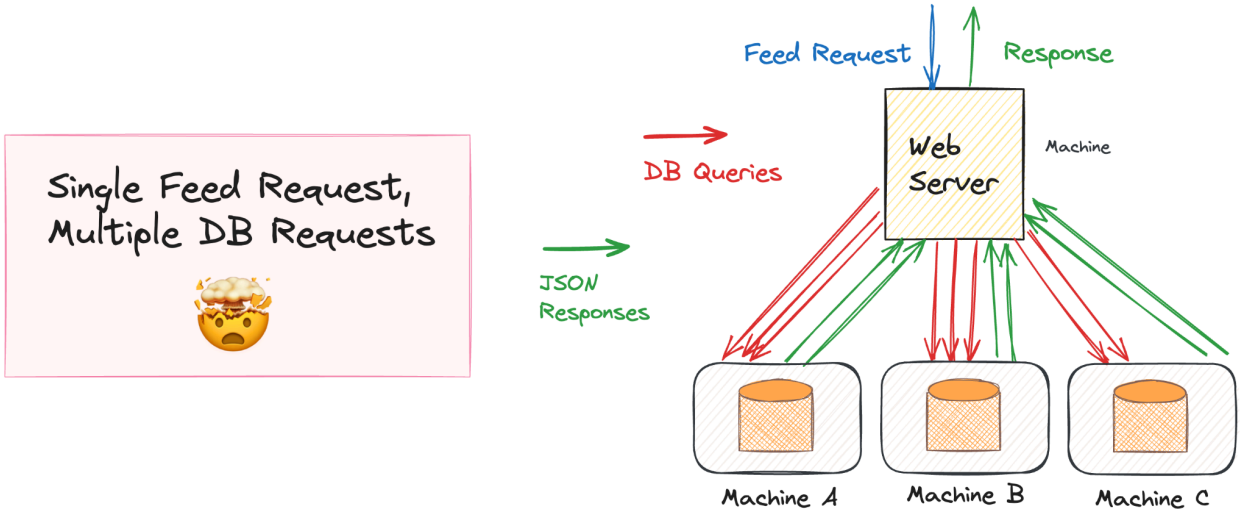
```
"data": [
  {
    "id": "post1",
    "type": "status",
    "created_time": "2024-05-17T12:34:56+0000",
    "message": "Had a great time at the beach today!",
    "from": {
      "name": "John Doe",
      "id": "user1"
    }
  },
  {
    "id": "post2",
    "type": "photo",
    "created_time": "2024-05-17T10:20:30+0000",
    "message": "Check out this sunset!",
    "from": {
      "name": "Alice Johnson",
      "id": "user3"
    },
    "picture": "https://example.com/photos/sunset.jpg"
  },
  …
  ]
```

Now, the web server needs to fetch each post ID and potentially images and comments for each post.

To do this, the web server makes additional requests to the database for each post ID, as well as for associated images and comments.



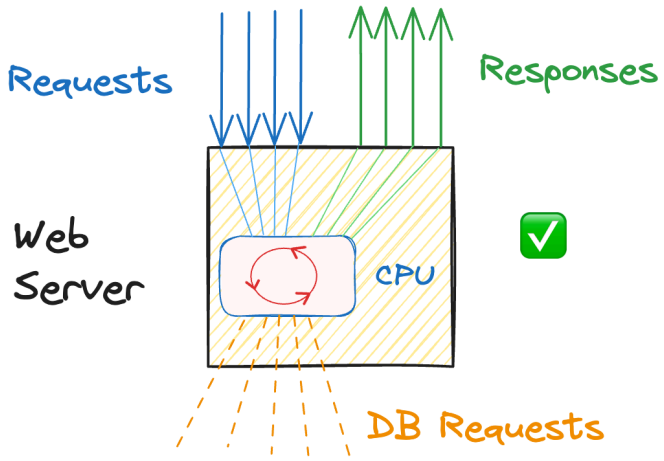
So really, it's not just one request; it's a multitude of database requests within one user request! Each user request can lead to many database queries, making the process more complex and time-consuming.



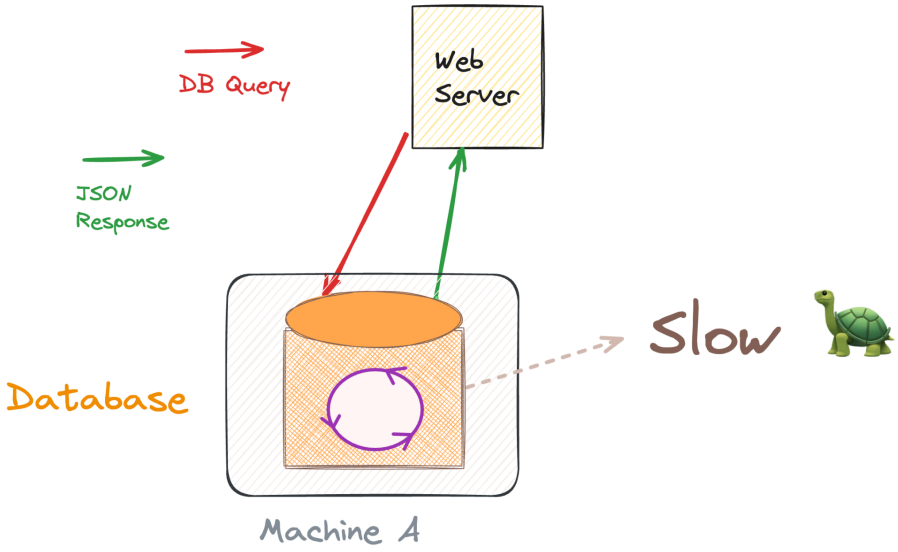
Don't believe me? This pattern is described in Facebook's paper "Scaling Memcached at Facebook."

Ok, so now we've seen the lifecycle of a request. Let's identify where the bottlenecks are. We can do a time analysis to see what takes the most time.

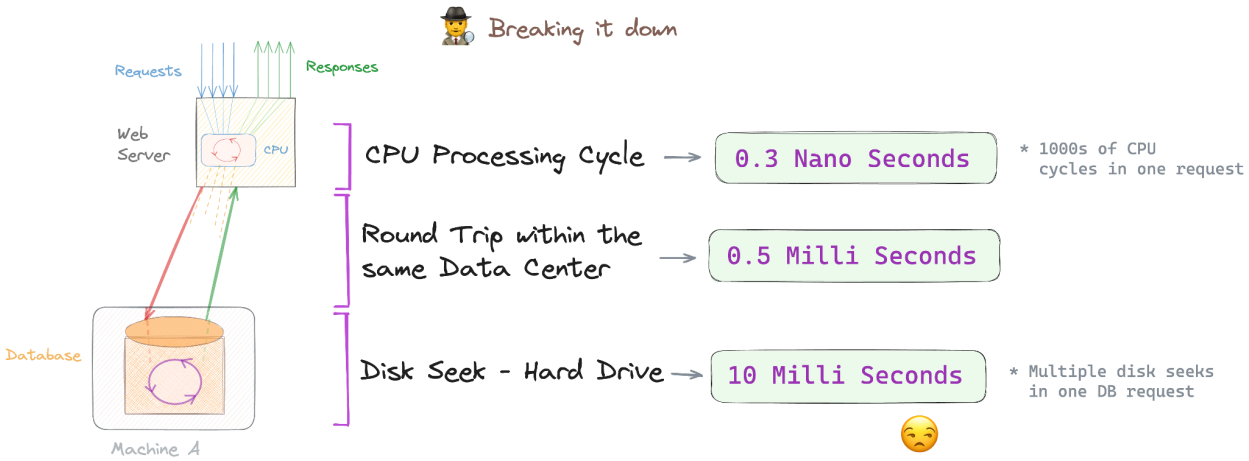
The request is processed in the web server, which uses CPU time. Assuming we have enough processing capacity, that should be fine. CPUs are fast.



But look at the round-trip requests the web server sends to the database. This part looks quite slow.



Let's compare the time taken by different components: CPU processing, database access, and network round trips. We can make a relative diagram showing which takes more time.



Let's look at the numbers.

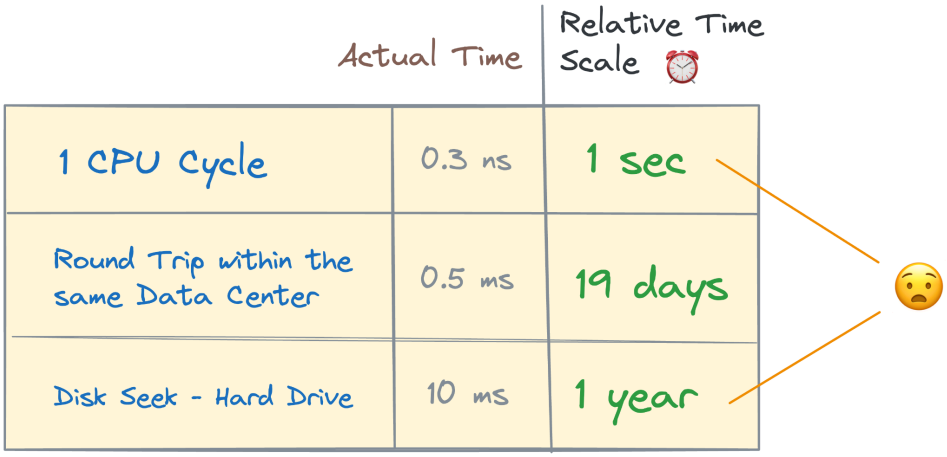
To memorize:

CPU Processing Cycle: 0.3 nanoseconds
Round Trip within the same data center: 0.5 milliseconds
Disk Seek: 10 milliseconds

Reference [1]

Disk seek takes 20 times as much time as a network round trip! This is where most of our time is going.

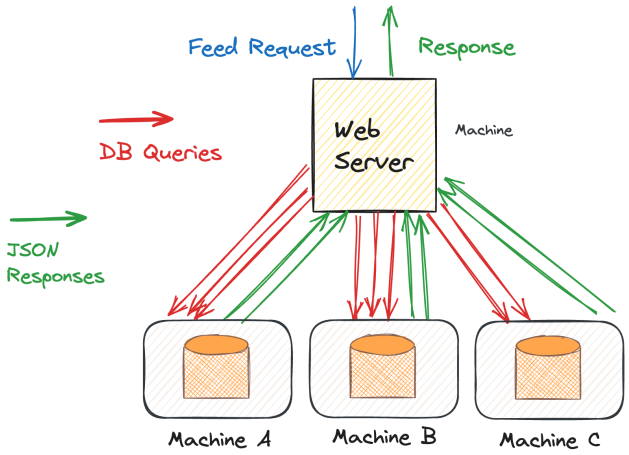
Now, nanoseconds and milliseconds are hard to imagine. Let's look at a scale which is easier to imagine for us humans. Twitter user @srigi converted them to human relatable values. Here is a simplified version of that:



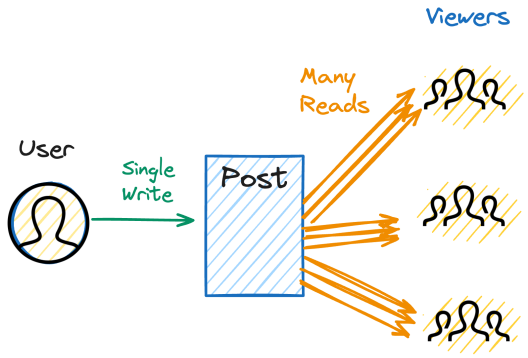
Now, do you see how slow disk seeks are?

The problem is worse than it looks. In reality, database requests take even longer. We only looked at disk seeks. We didn't consider the time taken by the database software or that each MySQL query might need multiple disk seeks, not just one. When you put it all together, we're looking at much more than 10 milliseconds!

And it gets even worse! We saw that each user request (like fetching a feed) takes multiple database queries. According to Facebook's paper we mentioned earlier, Facebook needed more than 50 data queries per news feed request!



Slow database reads are a huge problem because the vast majority of users are doing reads, not writes. There's a common saying: only 1% of internet users create content, and 99% read content—they're lurkers.



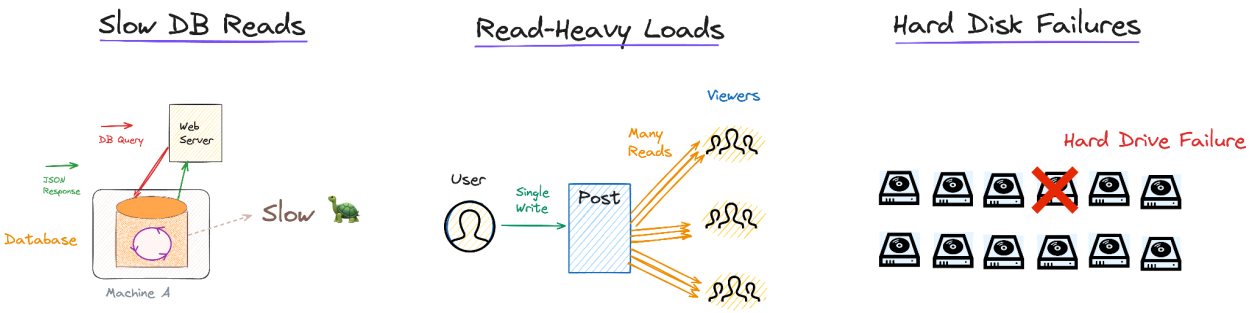
In a social network, while many users do create content (which is the whole point of a social network), the number of reads far outweighs the writes. Here's a simple way to think about it: for every piece of content created, there are multiple views, at least 100 on average. That's one write and 100 reads.

So yes, it seems like the database takes the longest time in our request process. It's the bottleneck.

There's another problem. It's not just speed. Disks rely on mechanical parts, which means the more you use them, the more they wear out and are prone to failure. Disks do fail. If you have a pool of 100 hard disks, it's almost certain you will see failures often.

The heavier the load you put on them, the more likely they are to fail.

We should really reduce our reliance on disks for these reasons. It seems like a good idea to look for alternatives.



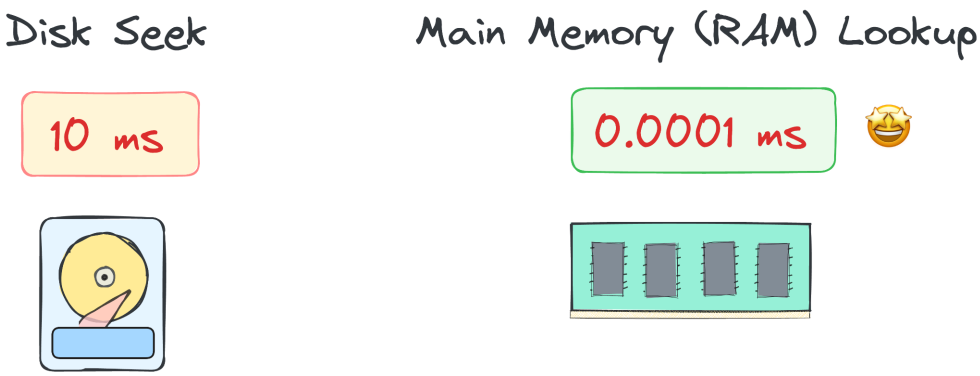
Ok, now that we know disks are a problem, how do we improve our system?

The Fix - how can we use RAM?

In the last section, we discussed the problems with our setup—specifically, the reliance on disks. While our system works, disks make it slow and unreliable. If we could avoid using disks, the system would be much faster.

So, how do we fix this?

To avoid disks, we need to store data in RAM.



To memorize:

Round Trip within the same data center: 0.5 milliseconds
Main Memory (RAM) lookup: 100 nanoseconds = 0.0001 milliseconds
Disk Seek: 10 milliseconds

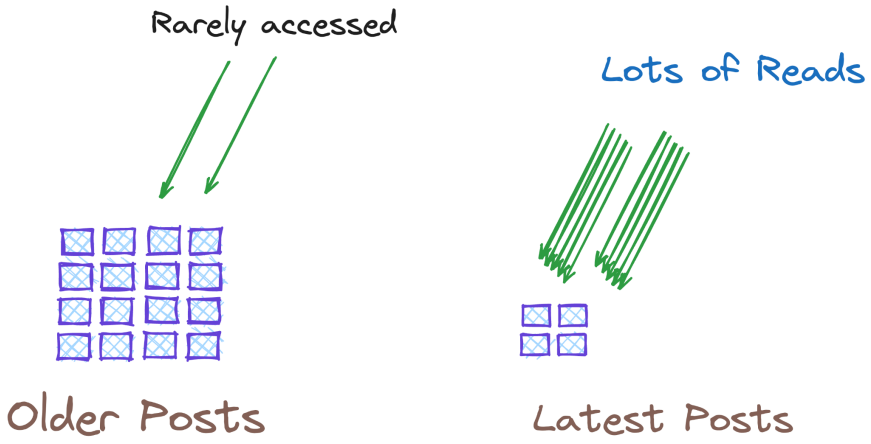
Reference [1]

RAM is 100,000 times faster than a disk seek!

Should we just replace the disk-based database? Well, we can't, because RAM is volatile memory—as soon as it is shut off, it loses data. So we can't rely solely on RAM.

Also, even if RAM could theoretically store data permanently, RAM is much more expensive than disk storage. Storing everything in RAM would be overkill, especially for data that isn't accessed often. For example, if we stored the entire user history in RAM, most of it would just sit there because, in a social network, only the latest information is accessed frequently. Older information is accessed much less often, and it's probably fine for those requests to take a bit longer.

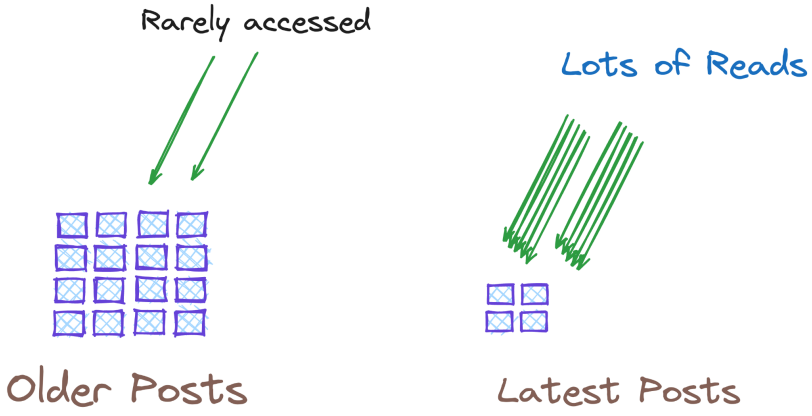
You can try this yourself. Scroll down your timeline and keep scrolling to posts from over a year ago. You'll notice that the load times become significantly longer. This is acceptable because you rarely make this request. How often do you look at posts from a year ago? Probably never, or close to never.



So that leads us to two observations:

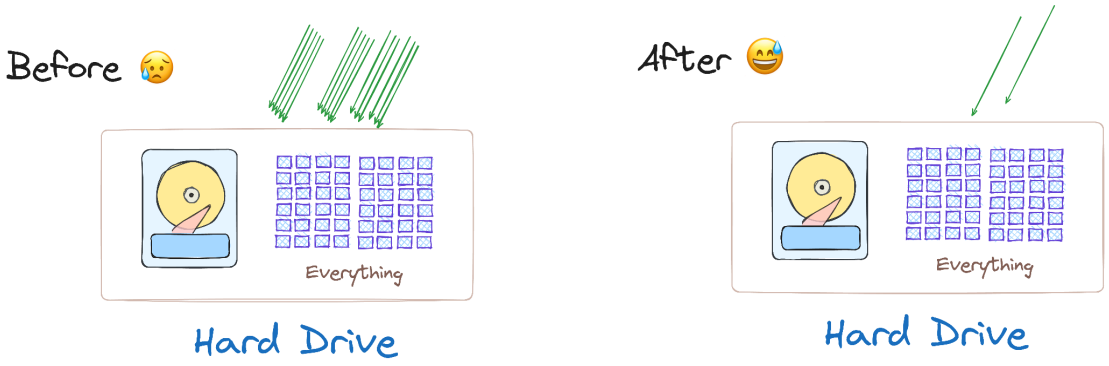
1. We can store the latest data in RAM for quick access, without needing to store the older data.
2. We can't rely on RAM to store data permanently since it's volatile.

This leads us to a natural solution: store everything in your hard drive-based database, and for the latest data, keep a copy in RAM. If this works, then a large majority of your read requests can be served by RAM and won't need to touch the database. For writes, we will still need to write to the database.

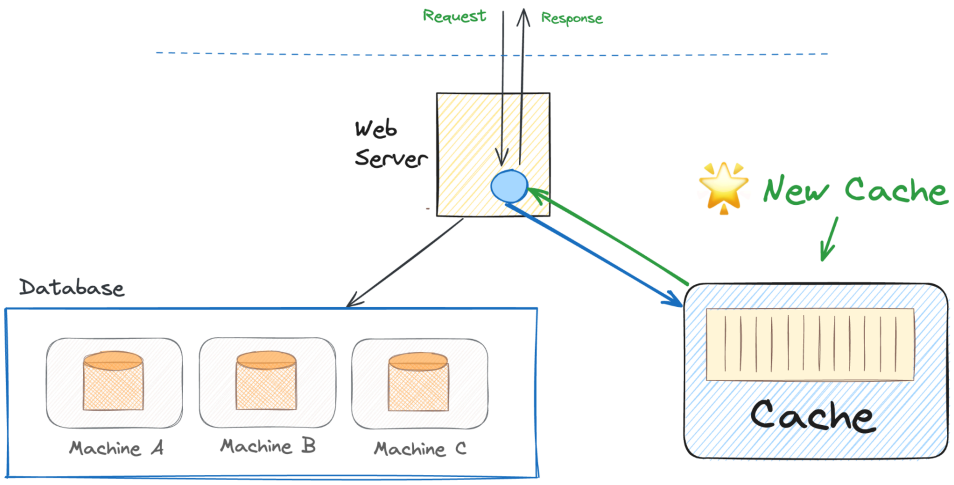


For a site like a social network, or most consumer sites, the number of reads far outweighs the number of writes. If we can serve reads faster for the user, that's a game

changer. For the DevOps team, this approach will greatly reduce the load on the database, which is also a significant advantage.



How do we implement this? We can add a separate machine, which serves as our RAM-based cache, or in-memory cache as we call it.



The Solution - How to use an Application Cache?

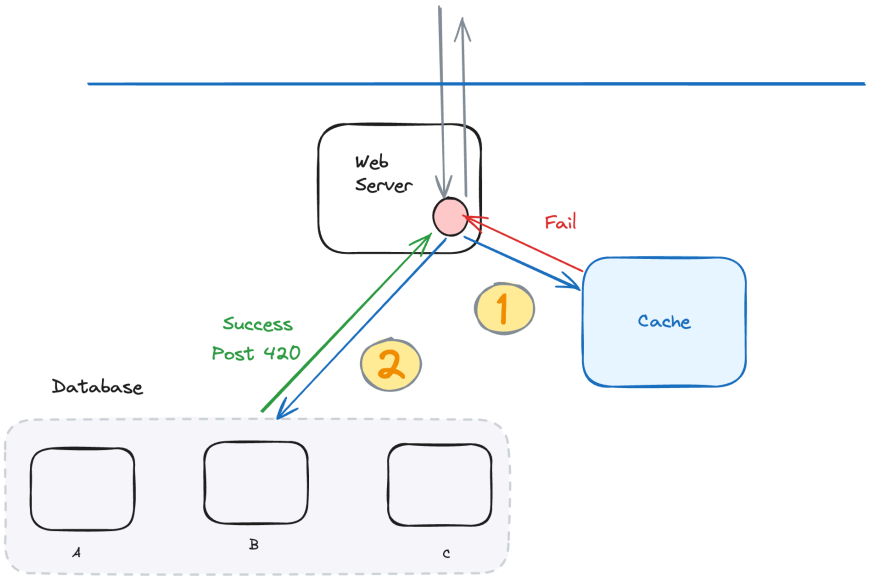
Let's take an example, yeah? Life is boring without examples.

Let's say we are fetching a post. Nice and simple. Something like - http://facebook.com/post/120 - 120 is the post id.

The web server says, "allright, I need to find post 420. If the cache has it, it will be fast. So let me check the cache first. If that doesn't work, I'll check the database."

The database will have it for sure, because it's job is to store everything. It's slow, yes, but it has everything.

So we come up with this flow for reads:



Ok, you might say, but now we're doing TWO requests. Earlier, we were doing ONE. What's with that?

Yes, that could be a downside of this read flow. But look, if we set up the cache right, we won't need the second DB lookup most of the time.

Also, remember our numbers?

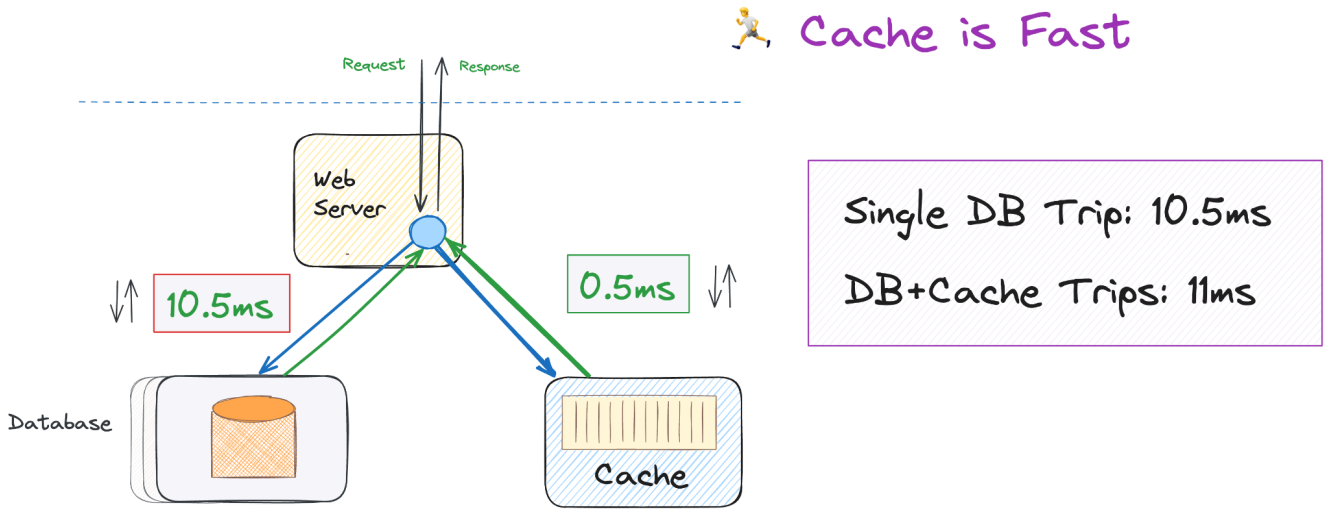
To memorize:

Round Trip within the same data center: 0.5 milliseconds
Main Memory (RAM) lookup: 100 nanoseconds = 0.0001 milliseconds
Disk Seek: 10 milliseconds

Reference [1]

This extra trip to the cache takes 0.5ms + 0.0001ms = 0.50001ms. Let's ignore the 0.0001, shall we? So It's just another round trip = 0.5ms.

The DB trip takes 10ms + 0.5ms = 10.5ms. So even if we take both trips, we're doing 10.5 + 0.5 = 11ms.



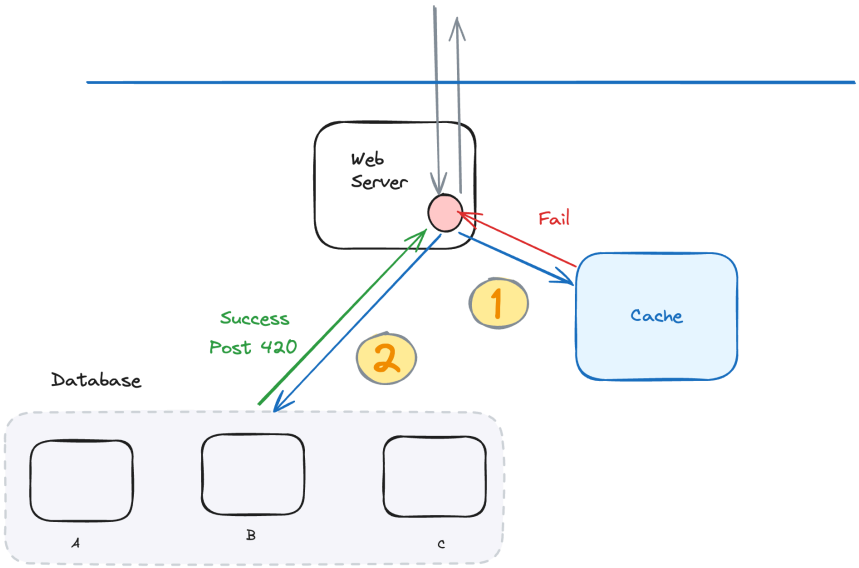
11ms instead of 10.5ms. That's not bad huh? So yea, this extra trip is not an issue.

Now, you might say, "hey Harsh, this is silly. You're not calculating things right. 0.5ms is just the network round-trip. What about the time taken by the cache to process the request? Or what about the time taken by the MySQL database to run the SQL query? Surely 10.5ms and 0.5ms are not the right numbers!!"

Well, you're right of course. 10.5 and 0.5 are not the actual numbers. However, they are a decent estimation of the relative time taken. Remember, the main point here is that disk is slow.

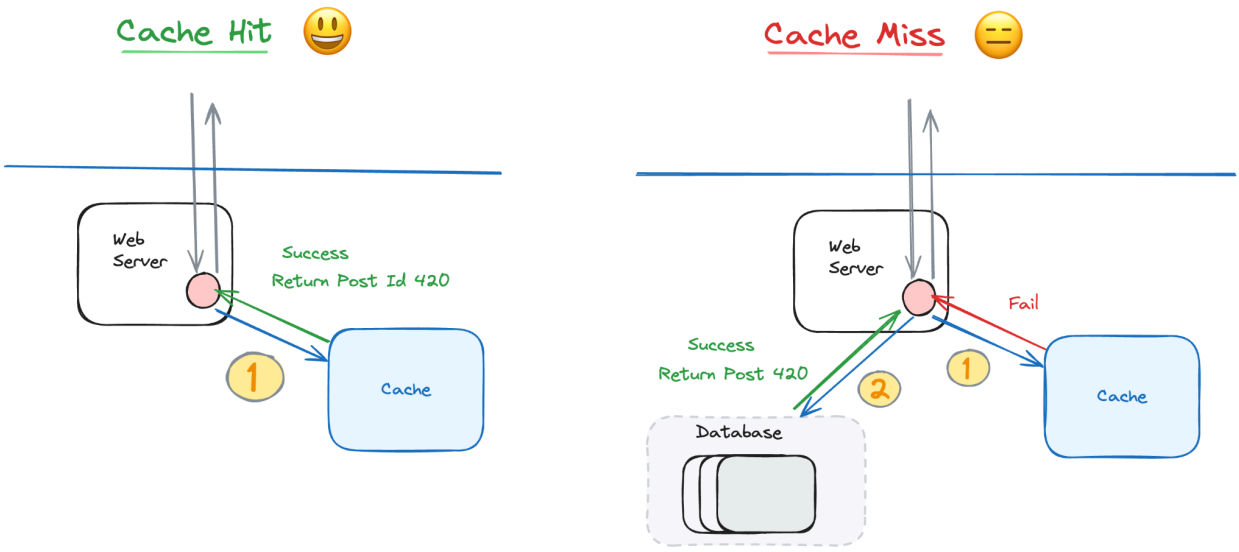
Those other things take CPU time, which is generally quite fast. They all pale in comparison to disk seeks!

Ok, we digressed. Here is the diagram again showing our read flow.



Remember, the web server checks the cache for post 420. If the cache has it, great, we return it and it's lightning fast.

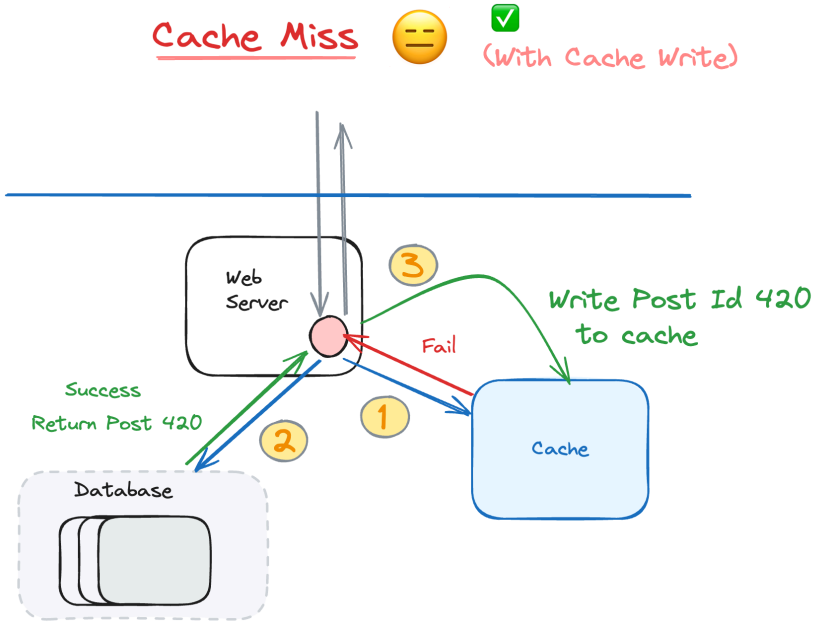
If not, we go through the database.



There's one more thing the web server needs to do after querying the DB. Can you tell?

It needs to write that Post 420 to the cache, so that it is available next time. Since it is being requested right now, there is a good likelihood that we'll request it again soon. That's just the nature of most data.

So we add a Step 3 to this - write our new shiny data to cache, so we don't hit the dreaded DB next time!



Oh, and this is called a "Cache-Aside" cache. You're keeping the cache "on the side", get it? It's part of the overall system, but loosely integrated.

This "Cache-Aside" is a "caching strategy".

There are other caching strategies, like "Write-through", "Read-through" and "Write-back". However, these other strategies are less used. Cache-Aside is king for most cases.

We'll go through other caching strategies later. It's good to know them for interviews.

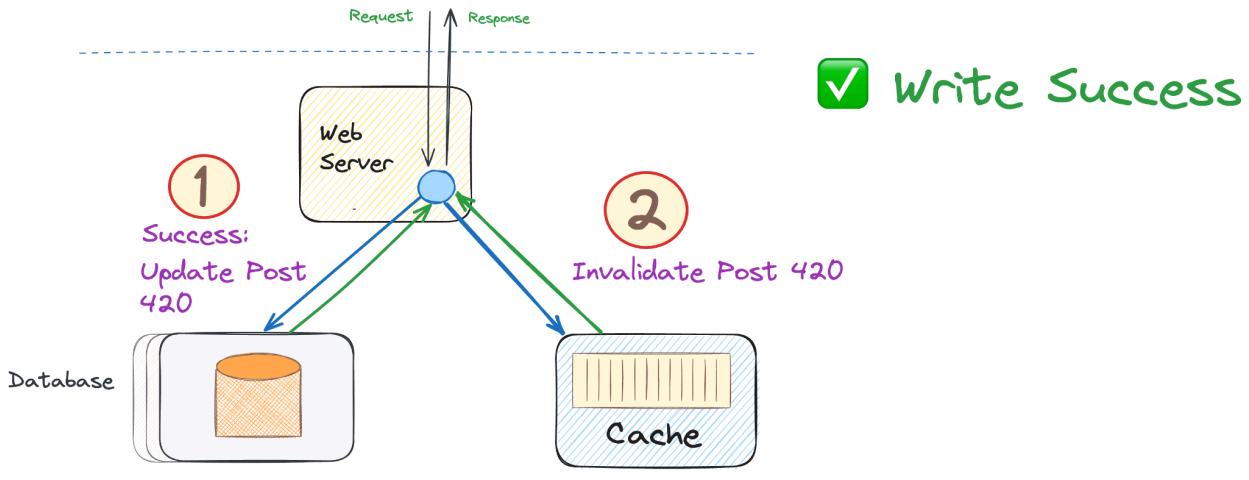
🤔 What happens inside the cache? We'll go into that in a bit.

OK, so far we've talked about reads. The next step is to talk about writes.

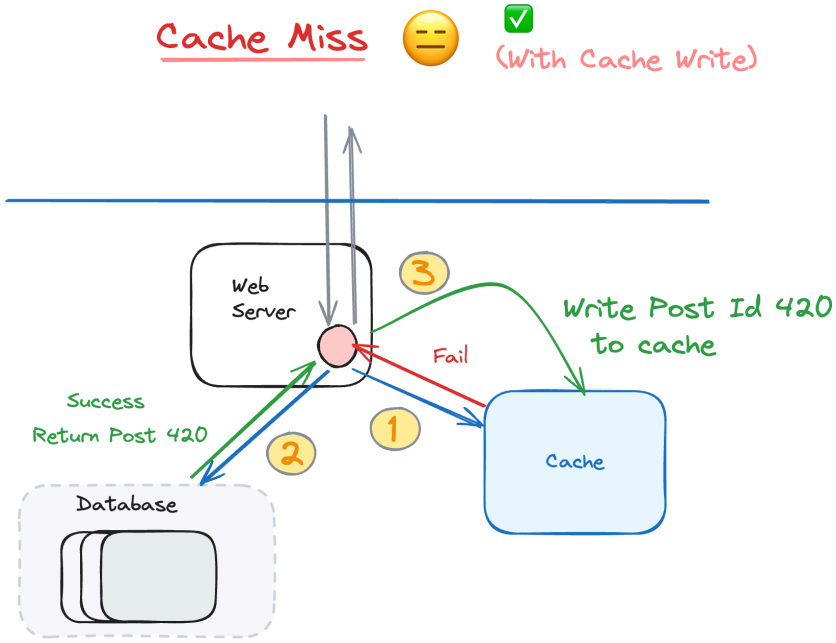
What about writes?

Let's talk about writes now. Users create content, and that content needs to be written somewhere.

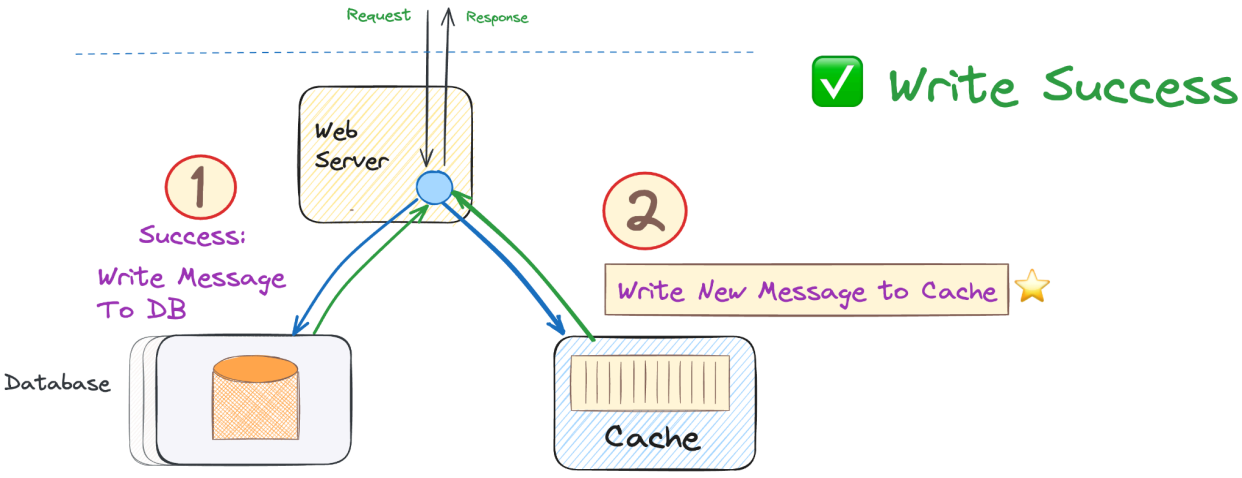
Writes have to go through the database—we don't have another option. When a write occurs, we write to the DB and then invalidate the corresponding cache entry so it will be fetched fresh next time.



The next time that entry is fetched, it will result in a cache miss because we invalidated it. The system will load the fresh data from the database. So yes, we will face one cache miss, but that's usually not a problem.



If we're really concerned about avoiding any cache misses, there is a way to do that. Instead of invalidating the cache, we could write the new value to it directly. This approach can be useful for certain types of data that we know will be accessed soon and need to be fast, like messages.



Why do we prefer invalidating over writing to the cache? Because invalidations are

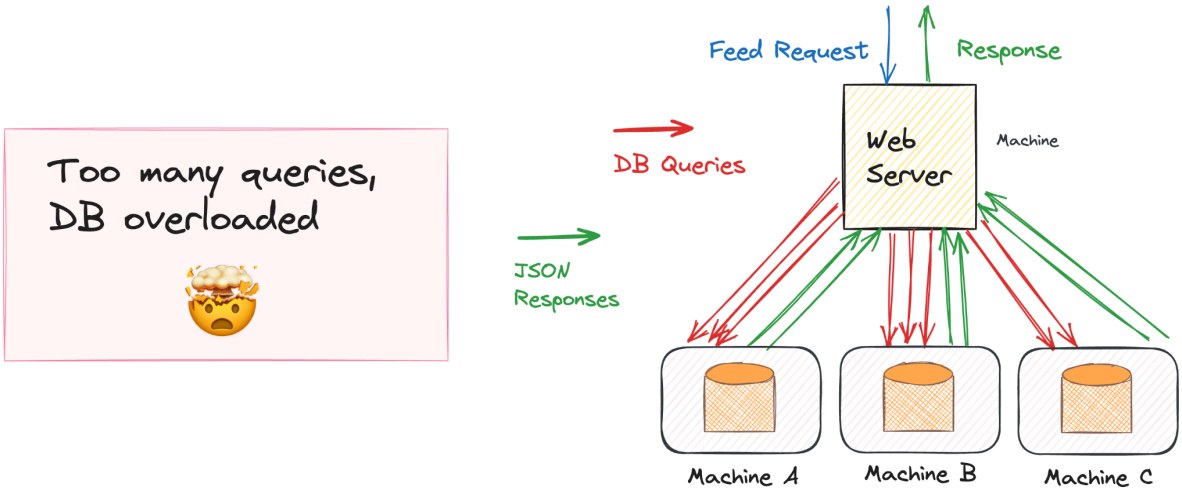
idempotent. This means that if there are multiple writes happening at the same time, it doesn't matter to the cache. There will be no race conditions or inconsistencies.

So, while it's possible to update the cache directly, invalidating the cache is generally safer and more reliable, ensuring data consistency without complications.

Ch 2. Caching Internals: How to build a cache from scratch?

We have realized that we need a cache for our social site. Great.

Now, we need to build a cache from scratch. Our system is growing, demand is increasing, and business is doing well. But we're noticing slow reads from our hard disk-based database, which is frustrating.



This is exactly what the creator of LiveJournal faced back in 2005. This is how memcached was born. Let's put ourselves in his shoes and build memcached from scratch.

Building cache on a single machine

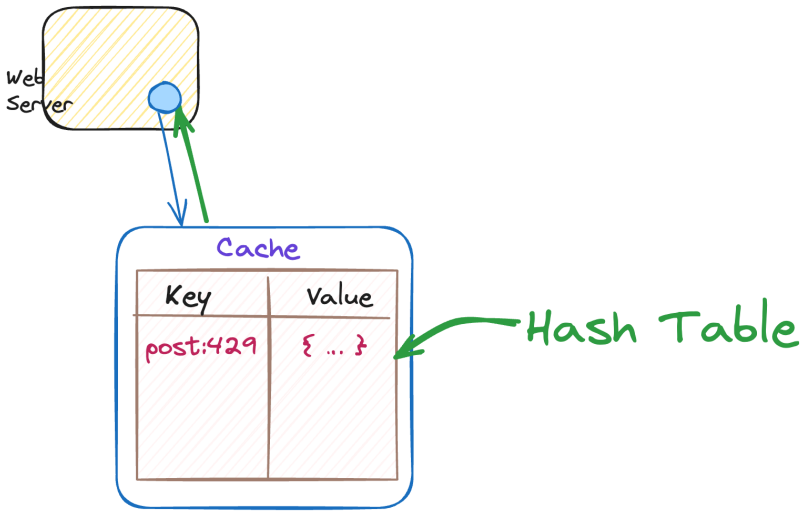
Let's improve our system by adding a cache machine. For now, we'll focus on adding just one machine.

A cache works like a key-value store. We put keys into a table and retrieve them when needed. It's pretty simple.

For example, if we want to cache post 429, we insert "post:429" into the table with the actual post content as the value.

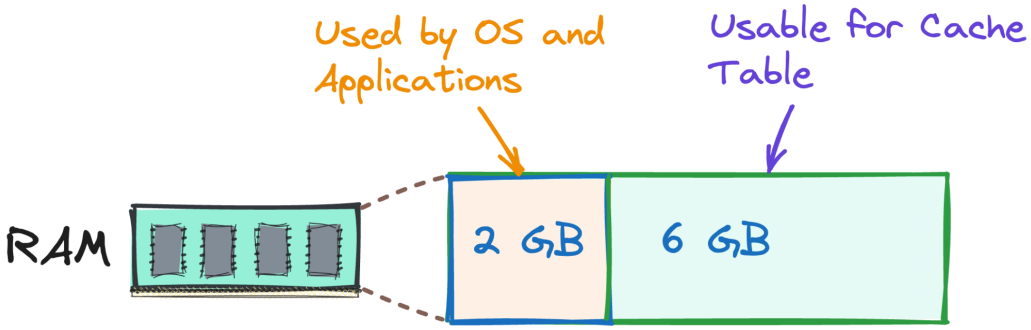
Imagine a table where one row has a post ID and the corresponding post in JSON format.

We need a hash table running on a single machine. To do this, we'll create a program that runs a hash table, which will be the base of our cache.

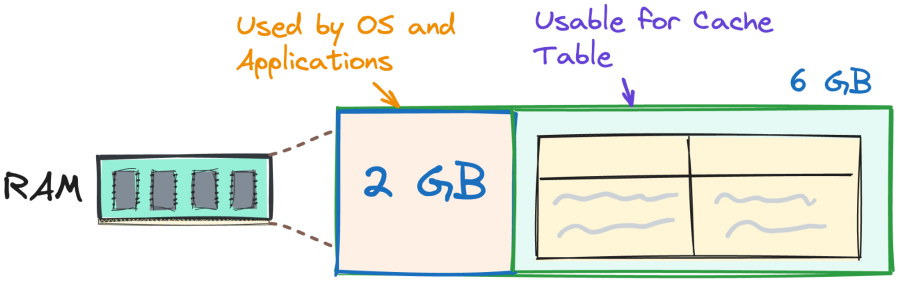


Let's say our machine has 8GB of RAM. The operating system and other applications will use part of this memory. After that, we have 6GB left for our cache.

- 8GB RAM
  - **OS and Other Applications**: 2GB
  - **Usable for Cache Table**: 6GB



In this 6GB of memory, we'll set up our hash table. This will be the maximum size of our cache.



We start by writing a simple program that can read from and write to a table. At first, this seems easy, but there's more to it.

Memcached is basically a hash table that communicates over a network. Let's break this down:

1. Local Hash Table: At first, our hash table can communicate locally, maybe accepting input from the command line.

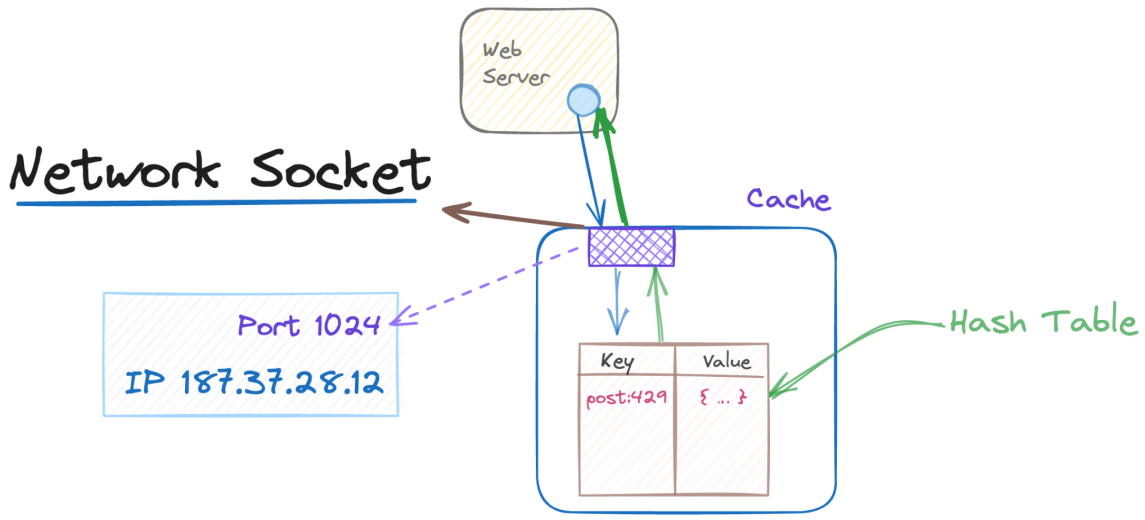
```
import sys
```

```
class SimpleMemcached:
    def __init__(self):
        self.store = {}

    def set(self, key, value):
        self.store[key] = value
        return "STORED"

    def get(self, key):
        if key in self.store:
            return self.store[key]
        else:
            return "NOT_FOUND"

    def delete(self, key):
        if key in self.store:
            del self.store[key]
            return "DELETED"
        else:
            return "NOT_FOUND"

def main():
    memcached = SimpleMemcached()

    while True:
        try:
            command = input("Enter command: ").strip().split()
            if not command:
                continue

            cmd = command[0].lower()

            if cmd == 'set' and len(command) == 3:
                key = command[1]
                value = command[2]
                print(memcached.set(key, value))

            elif cmd == 'get' and len(command) == 2:
                key = command[1]
                print(memcached.get(key))

            elif cmd == 'delete' and len(command) == 2:
                key = command[1]
                print(memcached.delete(key))
```

```
            elif cmd == 'exit':
                print("Exiting...")
                break

            else:
                print("ERROR: Invalid command or syntax")

        except KeyboardInterrupt:
            print("\nExiting...")
            break

if __name__ == "__main__":
    main()
```

2. Network Communication: Instead of using the command line, we switch to accepting input/output through a network socket.

To do this, we'll create a socket server in our program and open it to receive requests. Most programming languages have libraries to help with networking.



python

```
import socket
```

```
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(('localhost', 11211))
server_socket.listen(5)

while True:
    client_socket, address = server_socket.accept()
    command = client_socket.recv(1024)
    # Process the command
    client_socket.close()
```

With this setup, we can send commands to our memcached server using an IP address and socket.

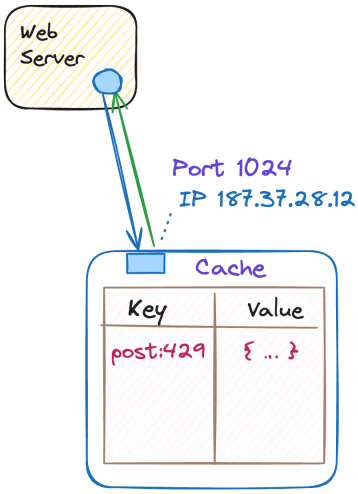
What commands can a web server send to our cache? The three main commands are:

- set(): Store a key-value pair.
- get(): Retrieve the value associated with a key.
- replace(): Replace the value of an existing key.

Here's how these commands might be used:

```
# Set a key-value pair
$ set post:429 {"title": "Post Title", "content": "Post content"}

# Get the value of a key
$ get post:429

# Replace the value of an existing key
$ replace post:429 {"title": "New Post Title", "content": "Updated
content"}
```

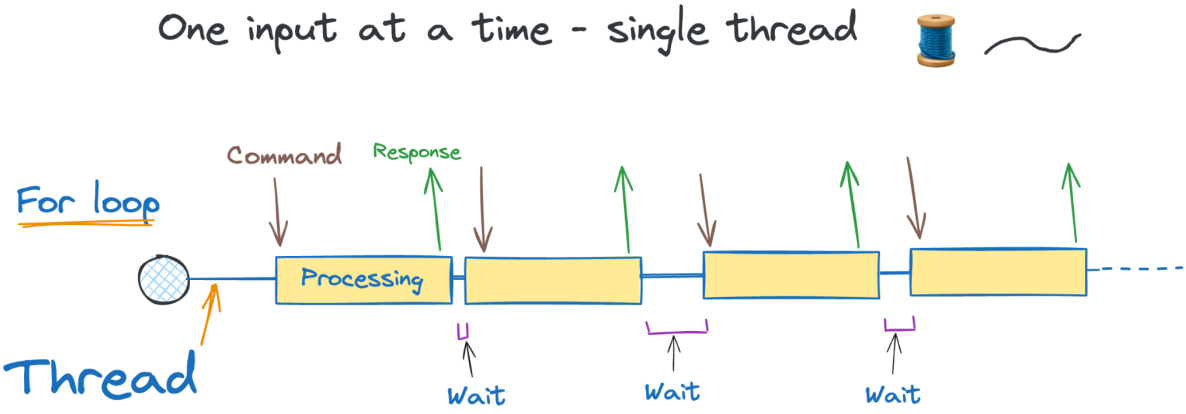
By following these steps, we've successfully built a working memcached server that accepts commands over the network. We've created this from scratch, improving our system with a powerful caching mechanism. Wonderful!



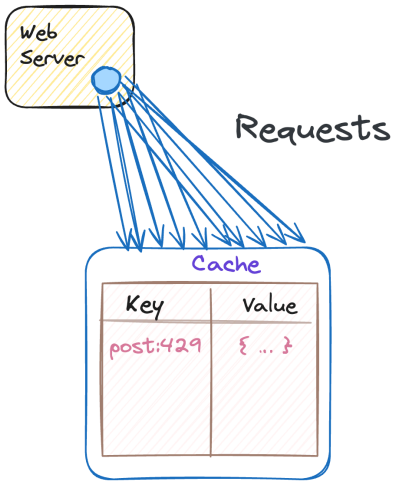
Do we need multiple threads?

Do we need multiple threads?

So far, we have built a memcached system with a single for loop. This loop takes one input at a time and gives one output. But in a real-life scenario, things are different.



A web server constantly sends requests to the cache.

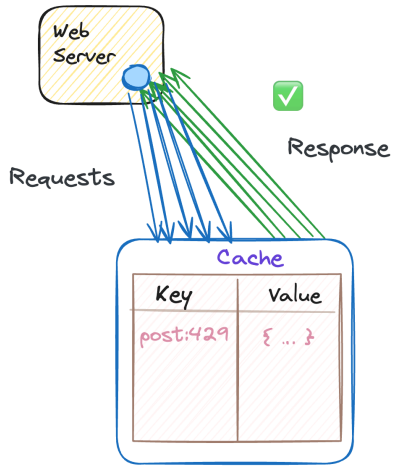


In fact, multiple web servers can constantly send requests. How do we handle that?

Let's first discuss handling a single web server connection. Imagine we have one web server connected to our memcache, and it sends requests very quickly.

Can our single machine handle the load? Yes, it can! Hash tables are incredibly fast to look up. In our system, the main thing we wait for is the hash table lookup, which happens from the main memory.

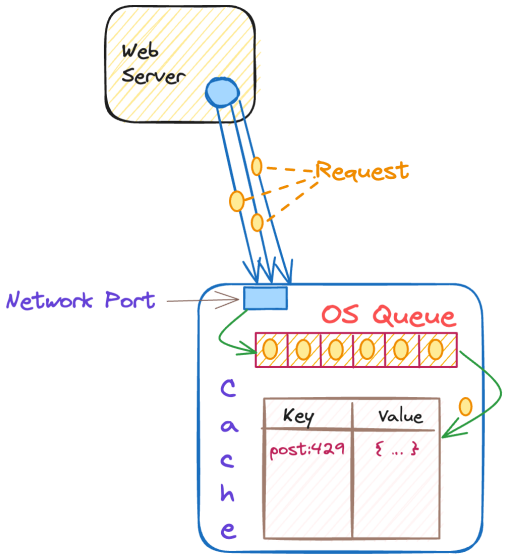
To understand this better, let's look at our latency table. Accessing the main memory takes approximately 100 nanoseconds. This means our system can handle about 10 million lookups per second! Given this speed, we don't need to worry about using multiple threads for this purpose. Your web server is not going to send you 10 million requests per second! The network can't handle that anyway!



If your network is indeed sending 10 million requests per second, either you are a billionaire already due to a highly successful service, or you're under a massive DDoS attack. Both of these situations are beyond the scope of this discussion!

So, we've determined that we don't need multiple threads if we have one web server sending requests. We can process each request so quickly that there's no need of parallelism.

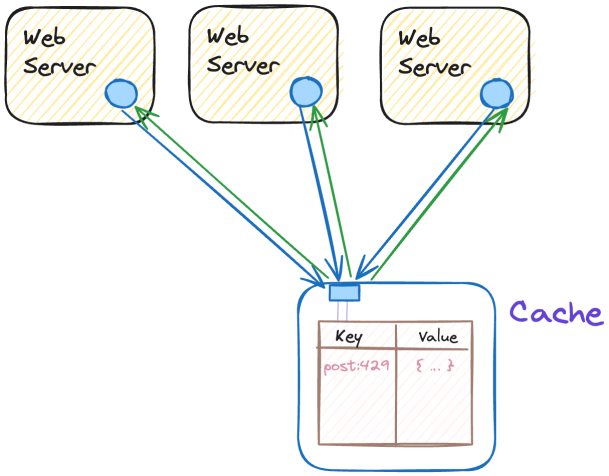
However, there might be times when multiple requests are queued up in our cache. For example, let's say that our system had an error or our OS paused for a moment. During this time, you might receive multiple requests. What happens then? Your operating system queues up these requests in a buffer. The networking software on your OS manages this queue, so you don't have to worry about losing any requests.



Your server program can simply accept requests one at a time. The operating system will handle the buffering, ensuring that each request is processed in order. This sequential handling works efficiently because the OS takes care of managing the incoming request queue, allowing our server to focus on processing each request quickly and efficiently.

Handling Multiple Web Servers

So far, we've seen that we're able to serve requests on our memcached to a single web server. But what about multiple web servers? What issues will we encounter there?

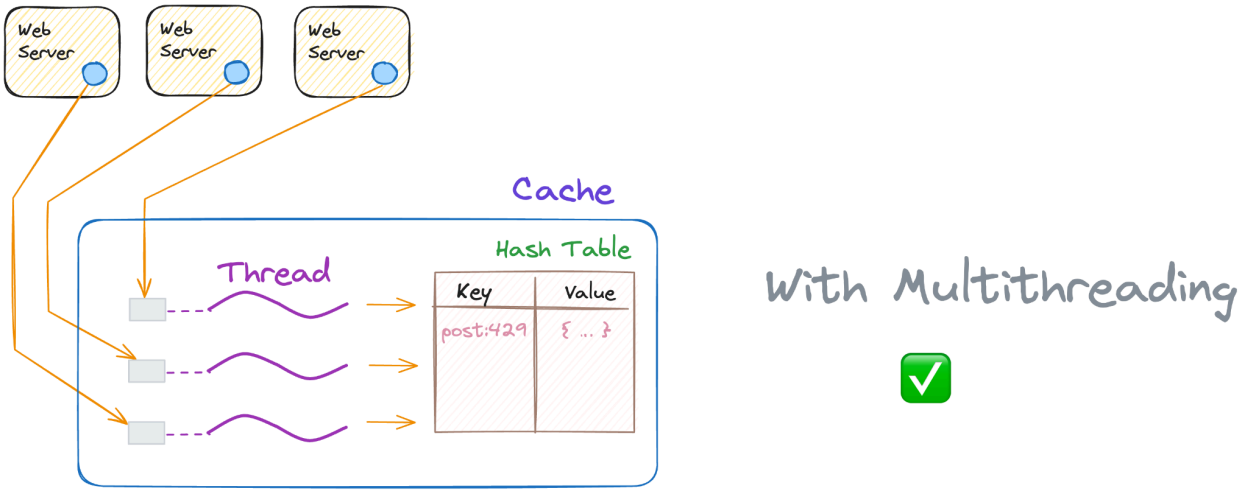


Let's say we accept server requests one at a time, in the following code:

```python
while True:
    accept a connection
    handle the request
```

We see that if one server doesn't send a request, our program will keep waiting for it, ignoring other server requests. If one web server dies or has a few seconds delay, we will keep waiting, and other servers will get queued up.

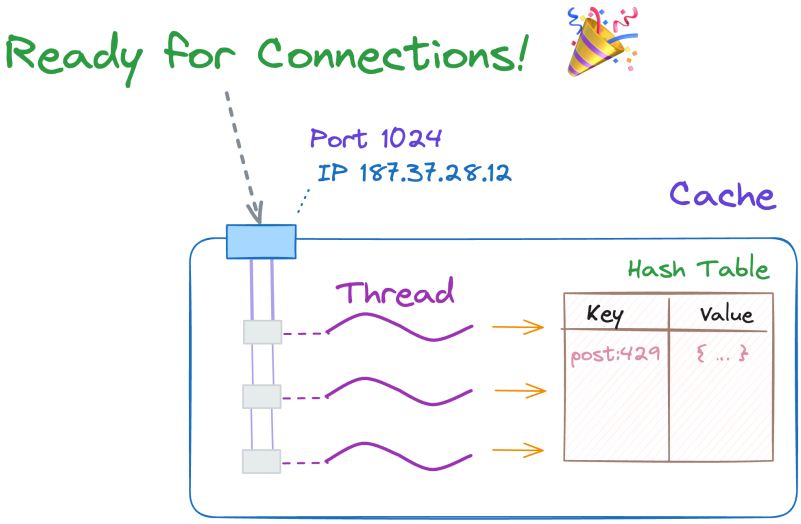
Luckily there is a simple solution to this: multithreading! For each connection request we get from a new client, we create a new thread. Each thread can wait and process the respective web server. Here is how it will look now:




```python
while True:
    accept a connection
    create a thread to deal with the client
```

And that's it! Voila! We are able to handle multiple web servers with our single machine memcached.

We now have a memcached ready for production! This would actually work if you decided to implement a memcached on your own!



Almost. There is one thing we haven't implemented here. Can you tell?

A note on few things we skipped, which will come later

We haven't discussed what happens when the cache is full. The way memcached handles this is by using an LRU (Least Recently Used) cache policy. When the cache is full, items that haven't been accessed for a long time get removed from the cache.

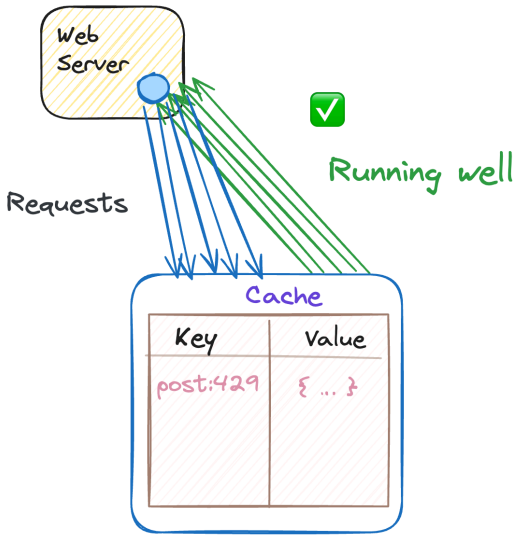
This way, a newer item has space. We'll discuss how this is implemented a bit later. For now, just note that this happens.

Ch 3. Distributed Cache: Scaling to Multiple Machines

When will you run into scale problems?

Let's review our setup so far.

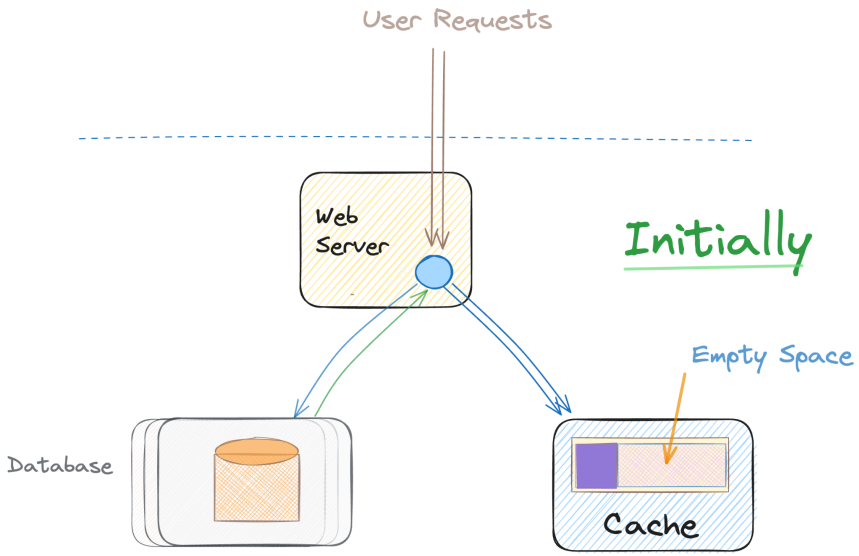
We have built a single machine cache by ourselves. Great. We added multithreading to connect to multiple web servers. We didn't need it for just one machine because it wouldn’t help. Everything is running smoothly, and we are serving our requests efficiently.

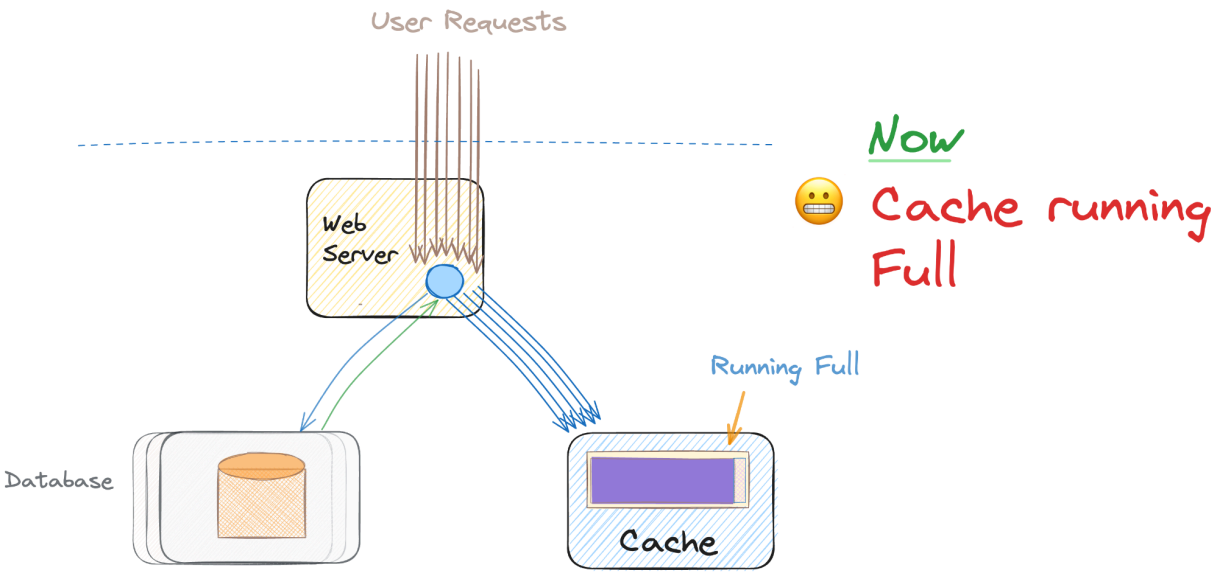


This is our current base setup.

Now, our site is growing, and we are getting more and more users. As our user base grows, so does the amount of data that is frequently accessed.

Imagine our data usage before and after the growth. Initially, we had a manageable amount of data, but now the amount has increased significantly.





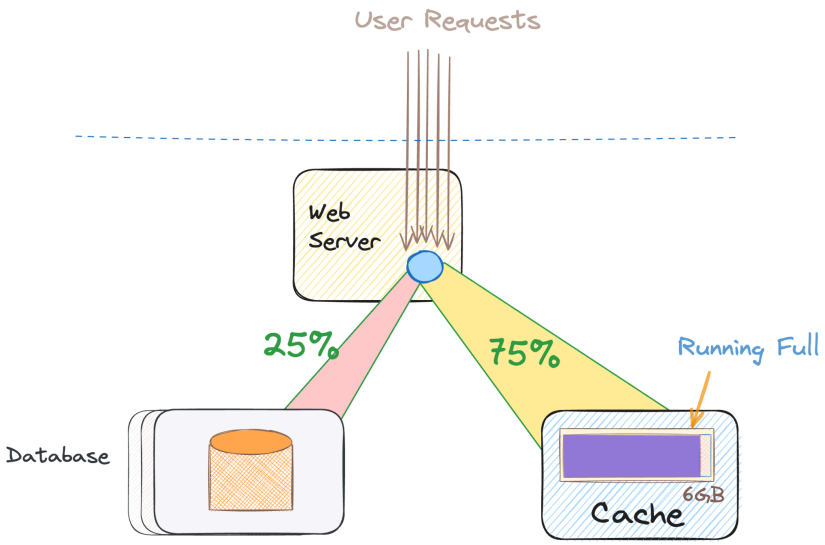
What does this mean for our system?

The amount of data in our cache increased. This data is frequently accessed from the cache. It's popular data that many users look at, not just one-off reads. For example, if a user posts a photo or text post, many of their friends, say 50 users, will view it. On a site like Reddit, a post might be accessed by thousands of users.

When the amount of frequently accessed data exceeds our cache capacity, we start facing issues. Let's say we now have an average of 8GB of data frequently accessed, but our cache can only hold 6GB. What happens then?

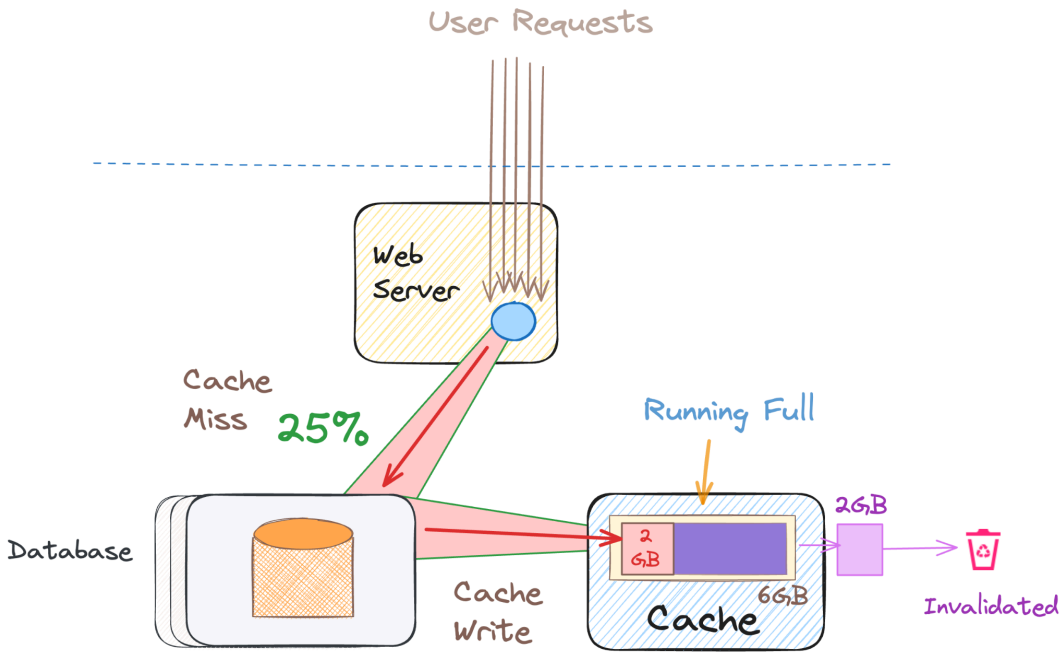
Two things will happen:

1. Approximately 2GB of requests (25%) will go directly to the database because only 6GB can be stored in the cache.
2. When a 2GB request is loaded from the database, it gets put into the cache, causing approximately 25% of requests to lead to cache evictions.



Imagine that, constantly evicting 25% of the data!

Since all 8GB of data is popular and frequently accessed, our system will constantly evict and reload data. This creates a chaotic scenario where the cache keeps replacing data while also sending many requests to the database. Not fun!

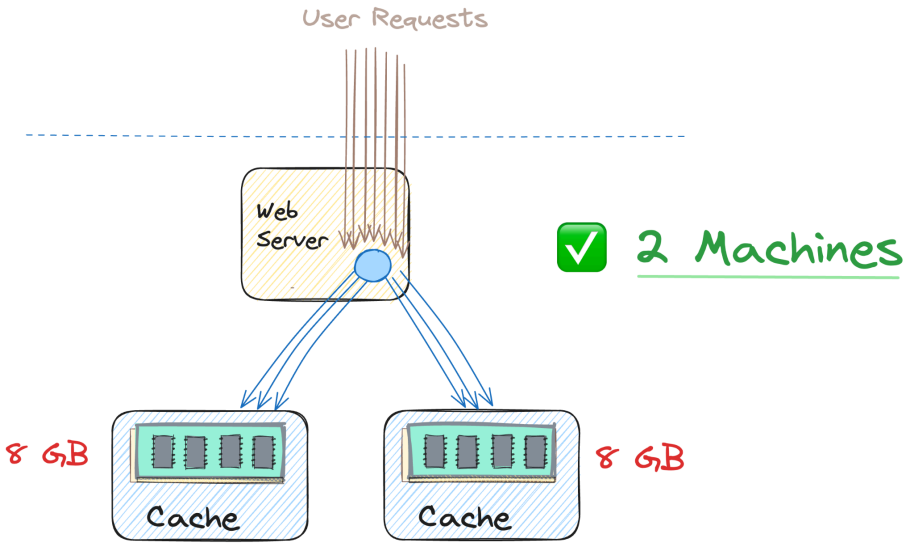


How do we fix this?

We need to add more memory. One way is to replace the RAM on our machine with a larger one. However, there are two problems:

1. RAM is expensive. The bigger the RAM, the more it costs.
2. This approach has limits. As our site continues to grow, we can't keep upgrading RAM indefinitely.

Instead, a standard solution is to add another cheap, commodity machine. Another machine with 8GB RAM will work, or even one with 4GB RAM.



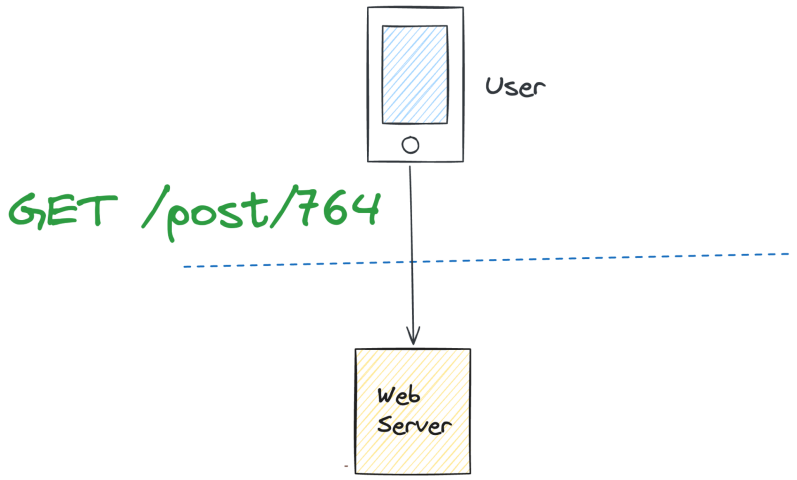
How can we integrate this into our current setup? We can use a simple modulo-based hashing on the server. If you're familiar with modulo-based hashing, that's great.

But before that, let's talk about how keys are queried from the client's side.

How does the web server interact with your cache?

For our cache's perspective, the web server is our "client". We use "client" and "Web server" interchangeably.

So we have some application code that needs to query memcached. Let's say we are running a fetch request for a post - www.facebook.com/post/764. Here is what our server code will look like:



python

```
# Example of server code making a request
post_id = 764
post_url = f"http://www.facebook.com/post/{post_id}"
response = fetch(post_url)
```

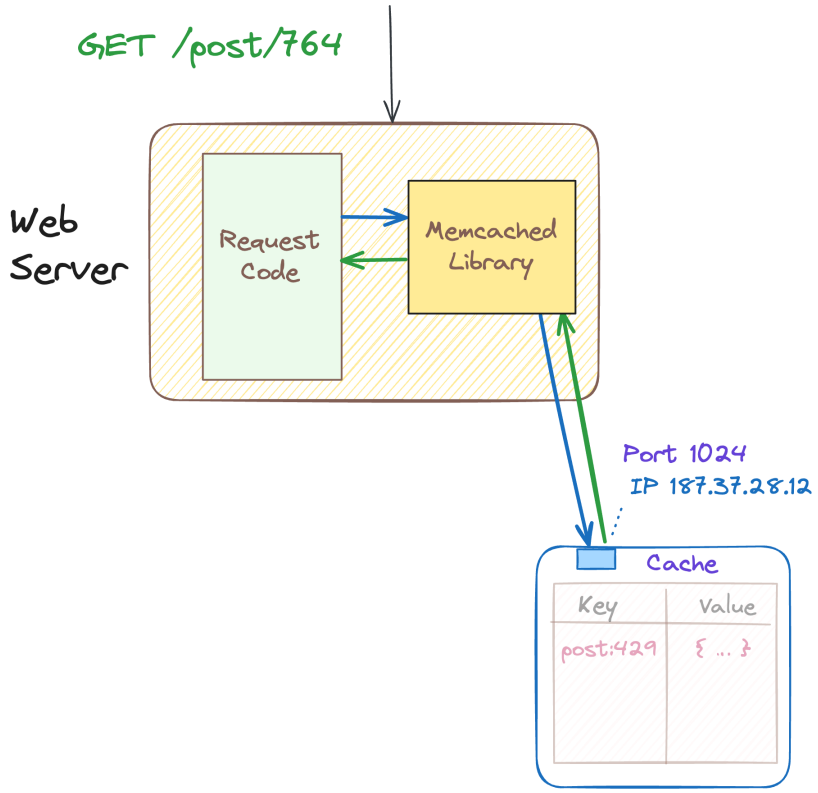
Now, we need to call memcached. This is done in the following way:

python

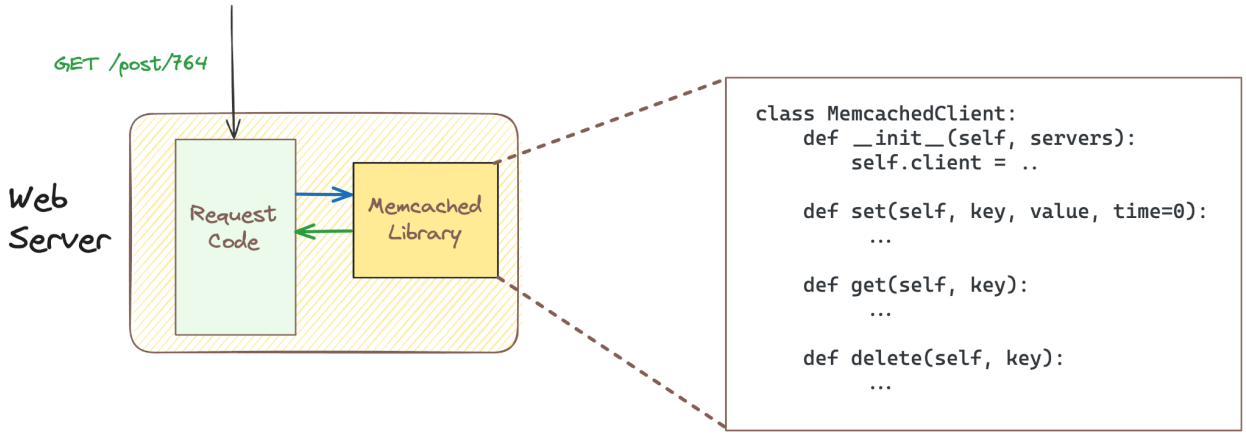
```
# Example of code using memcached client
import memcache

client = memcache.Client(['127.0.0.1:11211'])
post = client.get(f"post:{post_id}")
```

As we can see, there is a code library that we install with our web server so that we can use functions to call memcached. This is the memcached library. We use this to connect with memcached servers. How do we specify the servers? We give the IP addresses to the client logic.

Now, if we are implementing our own memcached, we can have a library on the web server. This library needs to know the IP address and Port of the memcached machines, so we give it that.



Now, the client library can connect and communicate with our memcached server. The web server code can use this library, and details of how the library interacts with the memcached are abstracted away.



Really, "client library" is a fancy term, it's just a separate set of functions wrapped in a package.

What if the memcached IP address changes?

In our current setup, if our memcached server needs to be reconfigured, or we change the machine IP, we will need to manually change the code, update the new IP and restart the memcached client object.

python

```
# config.py

# Old configuration
MEMCACHED_SERVERS = ["192.168.1.100:11211"]

# New configuration
MEMCACHED_SERVERS = ["52.23.198.192:2577"]
```

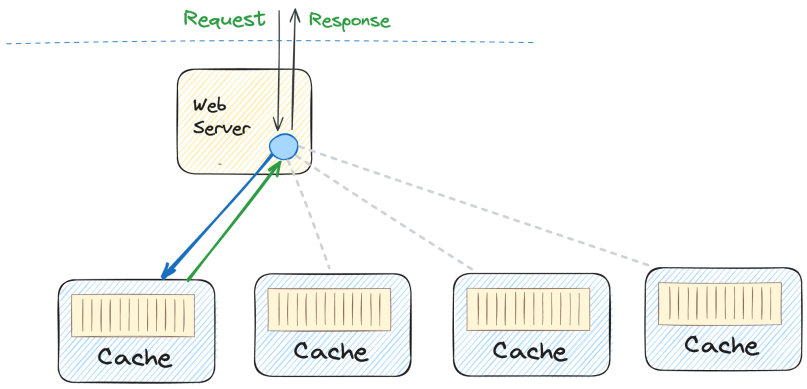
Ideally, we should be able to do it dynamically, i.e, auto detect a change in machine address and change it automatically in our client library.

But right now, there is no other way but to change the code. If we want to update the IP dynamically in the future, we can use a configuration service like Kubernetes, which automatically updates the client library with the latest IP address. This is more useful in large cloud setups though. In our tiny setup, it's probably overkill.

Scaling to 4, 10, and 100 more machines - the plan

Ok, so now we know how the client library works. Now, let's get back to our original problem. We need to add machines to our original setup. This was our original setup:

Let's say we want to add 3 more machines.



Let's do this in 3 steps. In each step, we make the design better.

Step 1: Simple Sharding (4 machines)
Step 2: Consistent Hashing (10 machines)
Step 3: Consistent Hashing with a Master (100 machines)

After each improvement, we'll see what improvement that step takes.

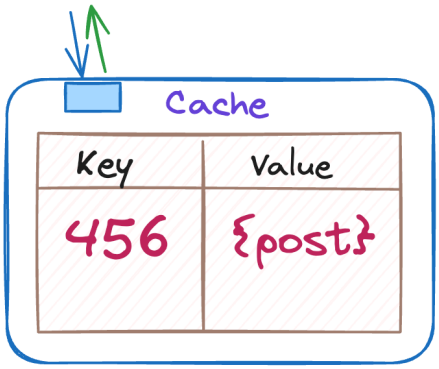
Ch 4. Scaling to 4 machines - Simple Sharding

So, we have our current setup. It's working well. Now, we need to add 3 more machines.

We plan to expand by adding 3 more machines to our site. This way, the web server will send requests to a total of 4 machines in the future. By spreading the load across 4 machines, we can handle more users and data.

Since we are splitting the load across 4 machines, we need to understand how the keys are accessed now.

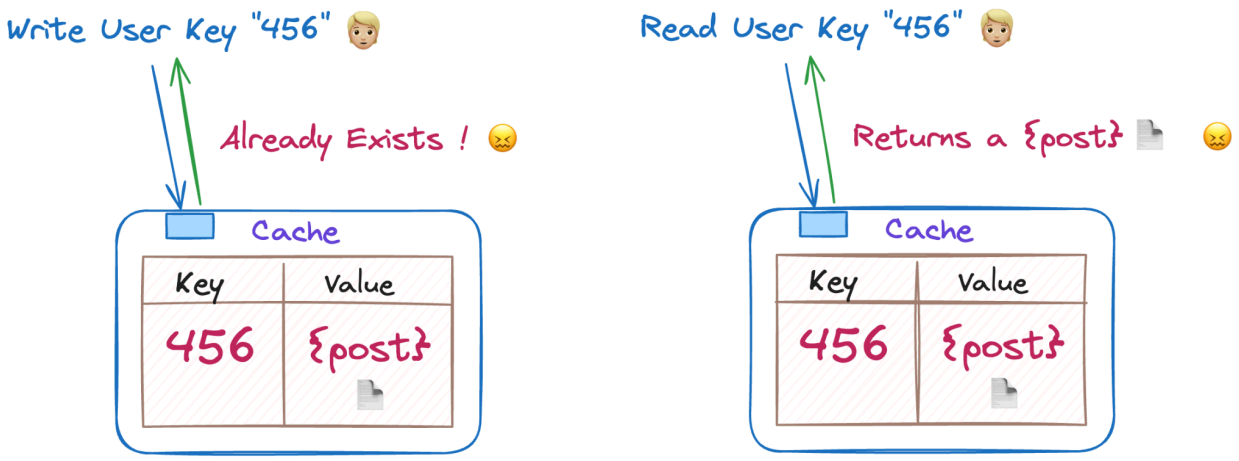
Right now, we use a hash table in the memcached server. For example, if we have a post with ID 456, it is stored in the cache with the key "456".



How to use Hash Functions for Sharding?

But, there is a problem here. Can you guess what it is?

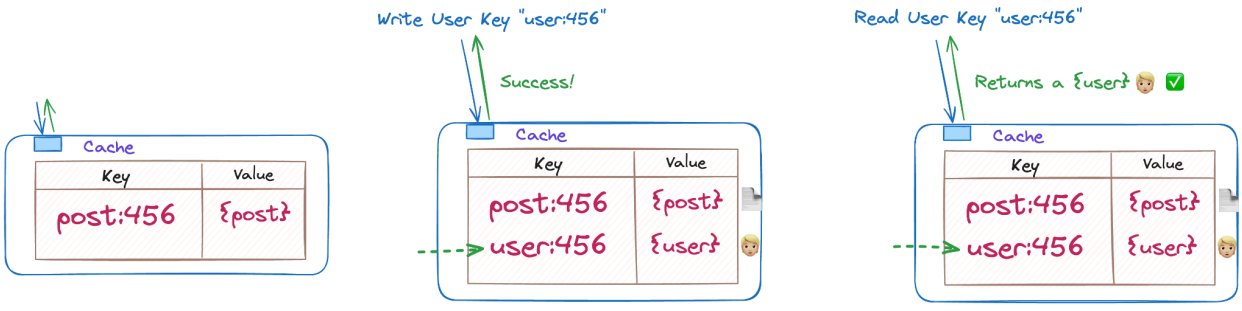
If we have a user with the ID "456" and we need to cache it, there will be a conflict. The key "456" for user and post will clash. The cache won't know if "456" refers to the user or the post.



Unlike a database, a cache doesn't have the concept of tables. There is no "post" table and no "user" table. In a cache, it's just one big hash table - all keys and values are stored together.

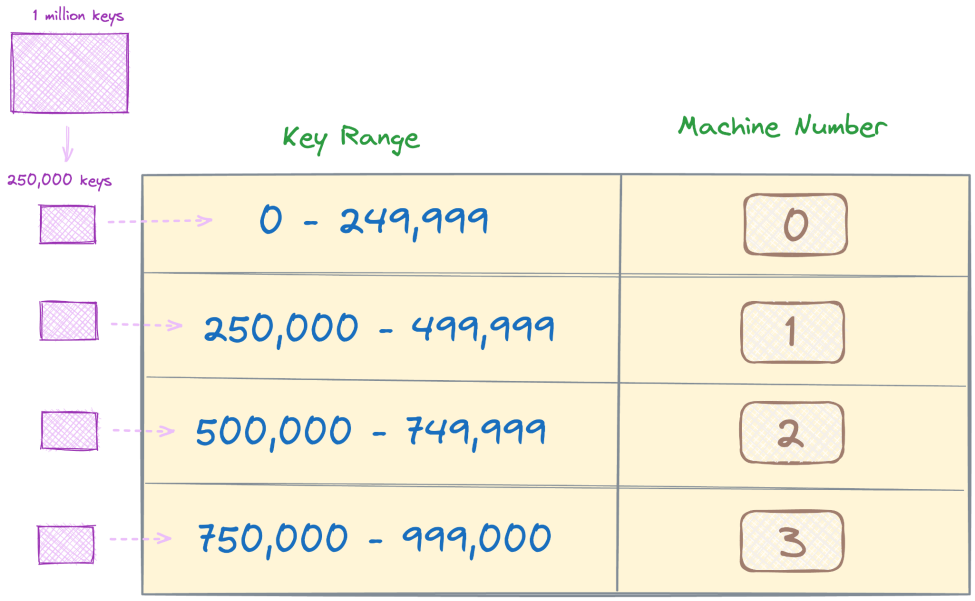
How do we solve this? A common way is by adding a prefix to the ID. For example, we store "post:456" as the key for a post and "user:456" as the key for a user. The colon (:) is a commonly used separator in both Redis and Memcached. This way, the keys are unique and there is no conflict.

So now we have:

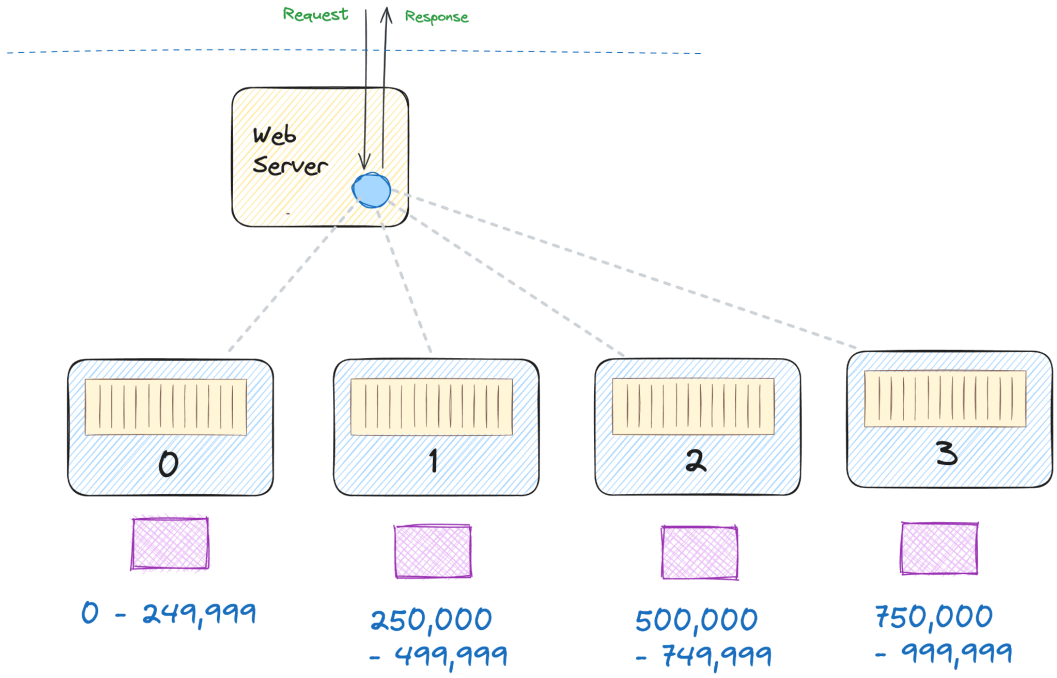


Now that we know this is one global hash table, how do we split it into 4 machines? One way is to divide the keys into 4 parts.

For example, if the key space of the hash table is from 0 to 1 million, we can assign 250,000 keys to each machine. So, machine 1 gets keys from 0 to 249,999, machine 2 gets keys from 250,000 to 499,999, and so on. This way, each machine handles a portion of the total keys.



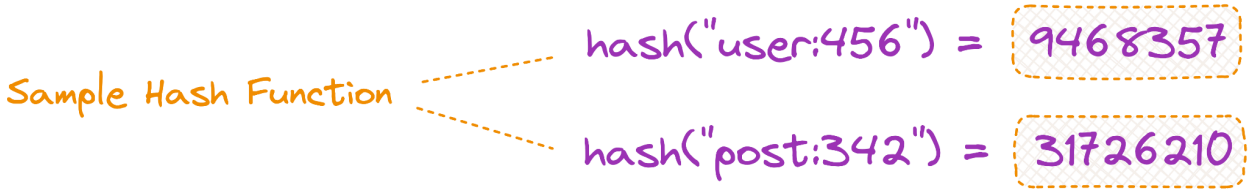
This is called sharding of the keys. Sharding means dividing a large dataset into smaller parts, making it easier to manage.



However, the client doesn't know anything about numerical keys.

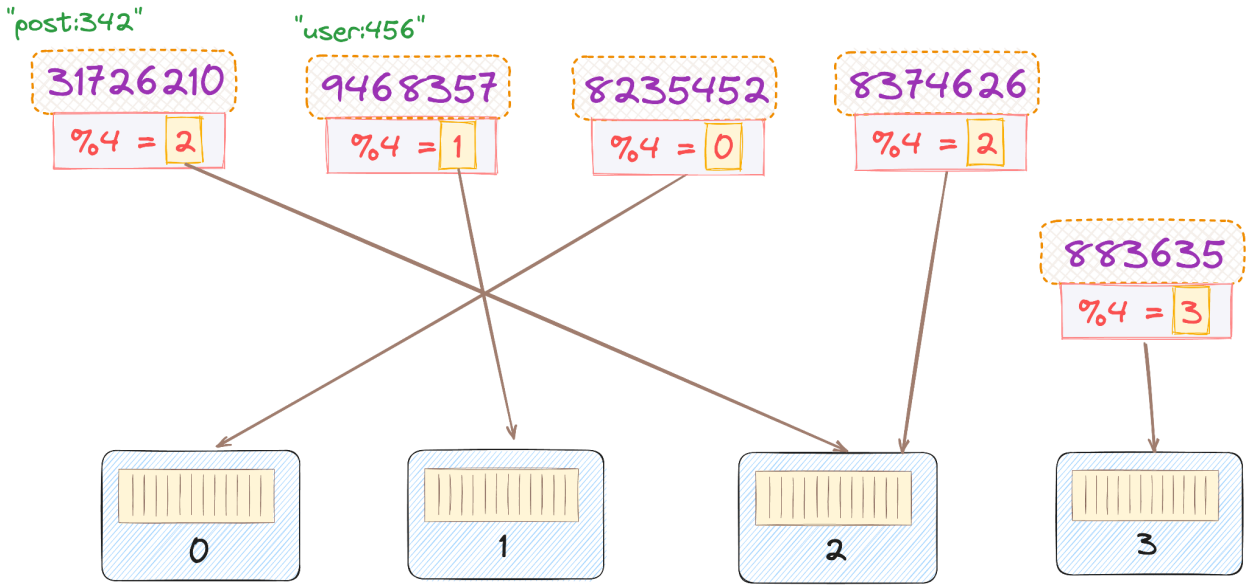
The numerical keys are part of the internal hash table. The client only sees string keys, like "post:456". There's no way for the client to map that directly to a hash number or a specific machine. So, here's what we can do: hash the string key on the client side using our memcached client library.

This way, every key, like "user:456", is converted to a number, like 947293472. This number is predictable because hash codes are consistent. The same value will always produce the same hash.



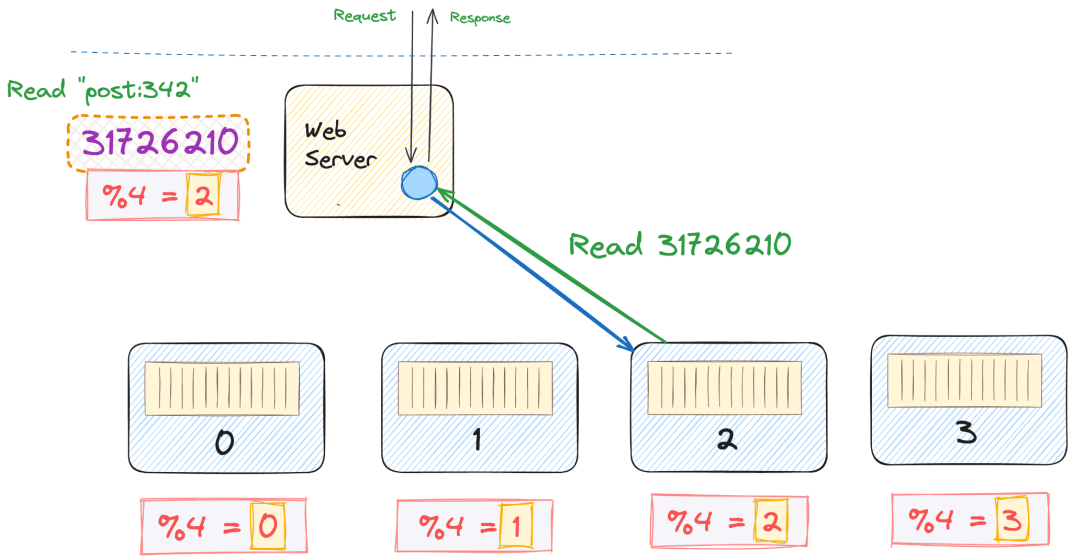
Now, we can use a simple trick to distribute keys across machines: modulo (%).

For example, if we have 4 machines, we can take the hashed key and do key % 4. This operation will give us a number between 0 and 3, which corresponds to our machine number.



We can use the formula machine(key) = hash(key) % 4.

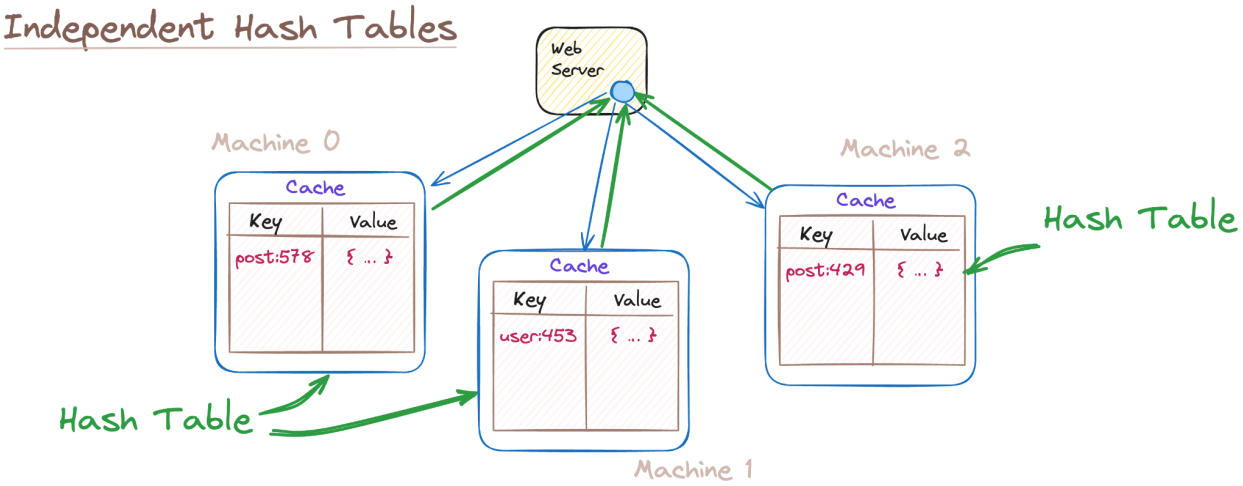
This is great because now we can send a key to a specific machine and retrieve it from the same machine.



Why do we hash the string? Good hash functions have two important properties:

- They evenly distribute keys, so one machine won't get overloaded.
- They produce the same hash for the same value every time.

Now, we don't even have to worry about splitting the hash table manually. Each hash table running on each machine is independent. It doesn't need to know about the other machines. It's simply an independent hash table.

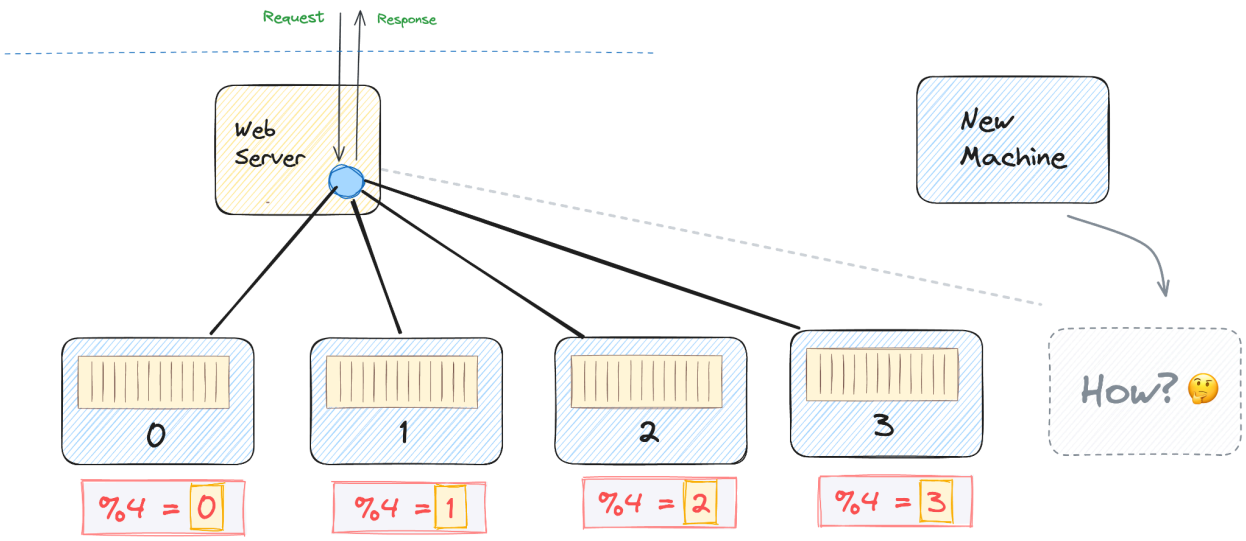


This method is called hash-based sharding. It is one of the simplest ways to shard data. And remember, with hash-based sharding, our keys are spread equally among all machines, so no machine gets unevenly overloaded!

Drawbacks of Hash-Based Sharding

However, this method isn't perfect for a few reasons.

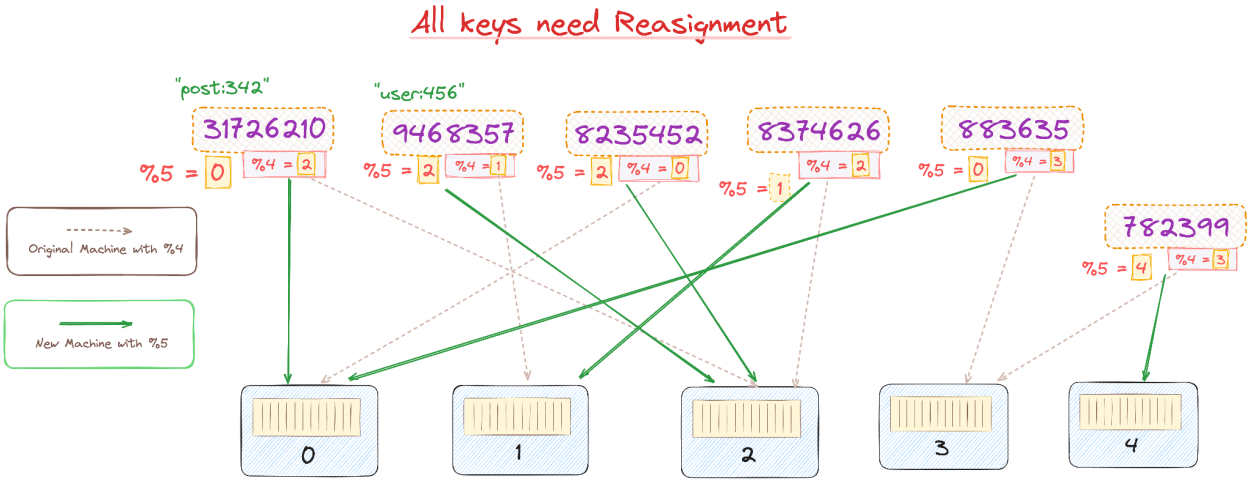
Let's say we need to add a new machine to our setup. How do we do that?



Our new machine function will have to change from %4 to %5 -> machine(key) = hash(key) % 5.

So, if a key like 78 was previously in machine 2 (78 % 4 = 2), it will now be in machine 3 (78 % 5 = 3). This change will affect all the keys in our cache, not just a few. Every key that was mapped to a machine using %4 will now have a new machine assignment using %5.

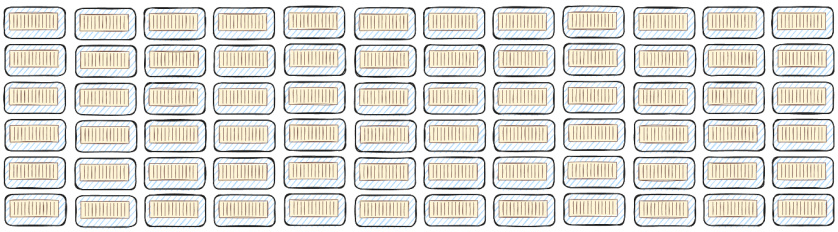
This means that all the keys need to be remapped and moved around before we can start serving requests with the 5th machine. This isn't a simple task; it involves a lot of data shuffling and coordination.



And here's the kicker: every time we want to add another machine, we will have to remap every key. This means we have to stop the entire cache system, take the time to remap all the keys to their new machines, update the modulo function in our code to reflect the new number of machines, and then restart the cache system to start serving requests again. It's like having to rearrange your entire house every time you buy a new piece of furniture! This process can be time-consuming and disruptive to the service.

This method means downtime for the user and a lot of administrative work for us, the engineers or cloud admins.

It might work fine for our small home server setup, but it's not practical for large-scale systems. Imagine running Facebook's backend with thousands of machines in your cache. You'd go bonkers trying to remap the whole system every time you added a new machine!

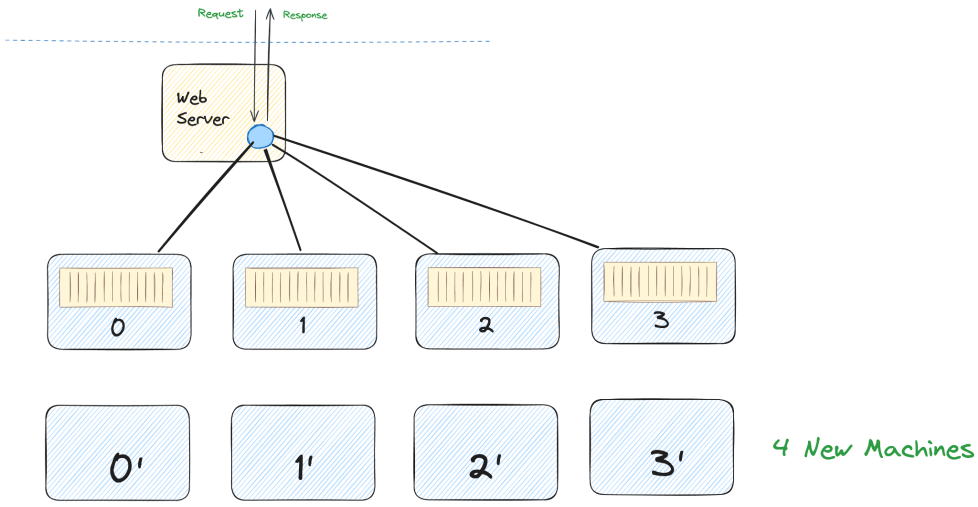


Now, while this method is not used in large scale systems, there are two workarounds for this problem that you should know about, especially for interviews. Here they are:

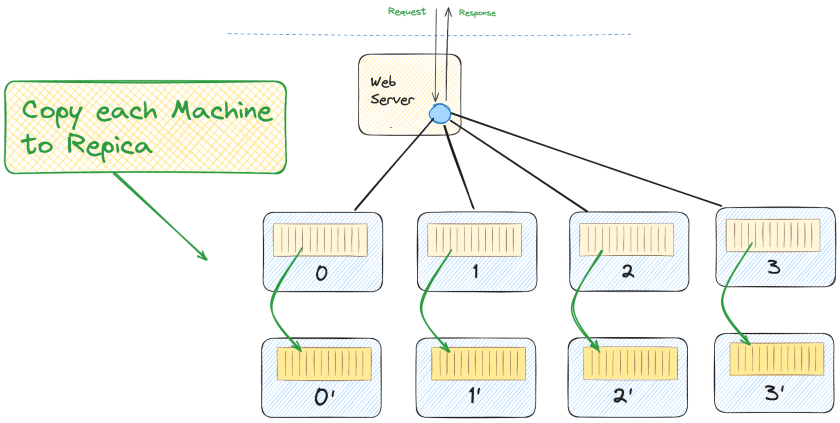
1. Using replicas while upgrading
2. Adding double the machines

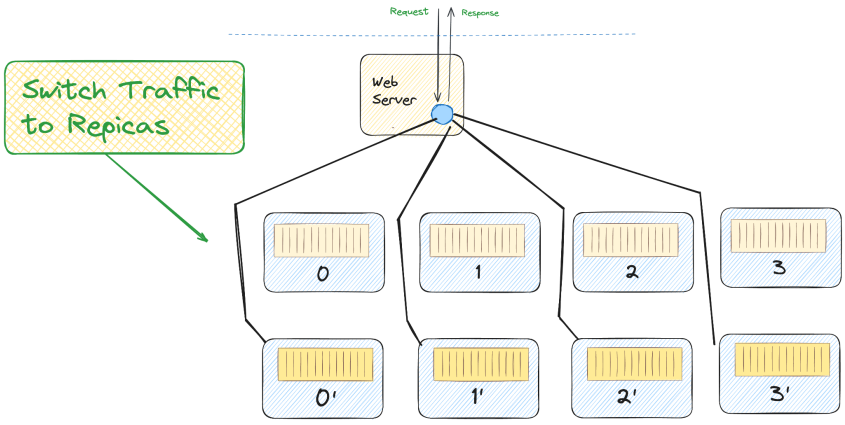
How can we use Replicas while Upgrading?

Let's talk about the first one. Normally, you would have to stop serving cache requests, remap all the keys, update the machine formula, and then restart the requests. Can you prevent stopping requests somehow? Yes, you can, but it would cost a bit more.

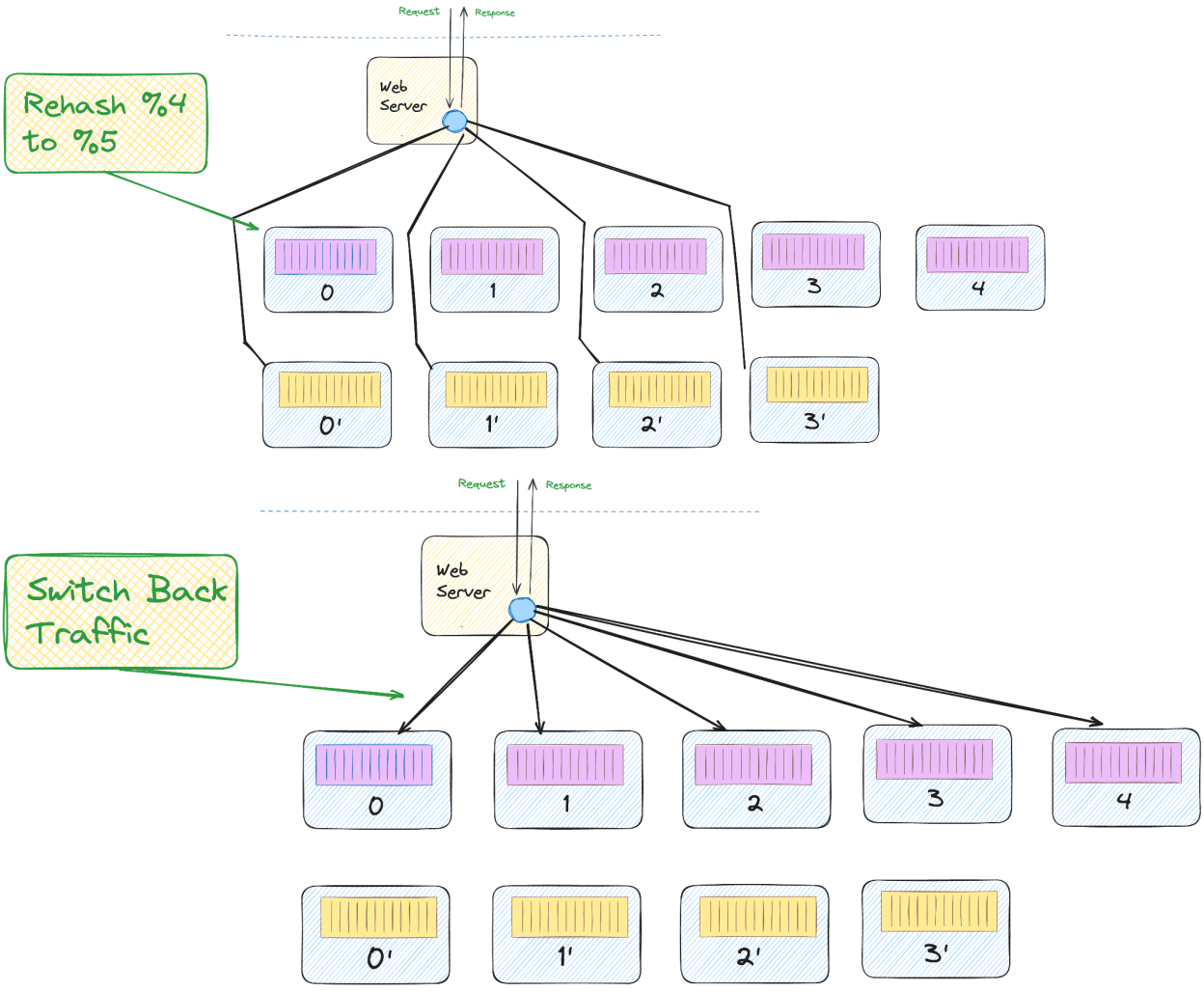


We can rent out 4 new machines and copy our existing cache to them. These new machines can serve traffic just like our original ones. This way, we avoid downtime and ensure continuous service.



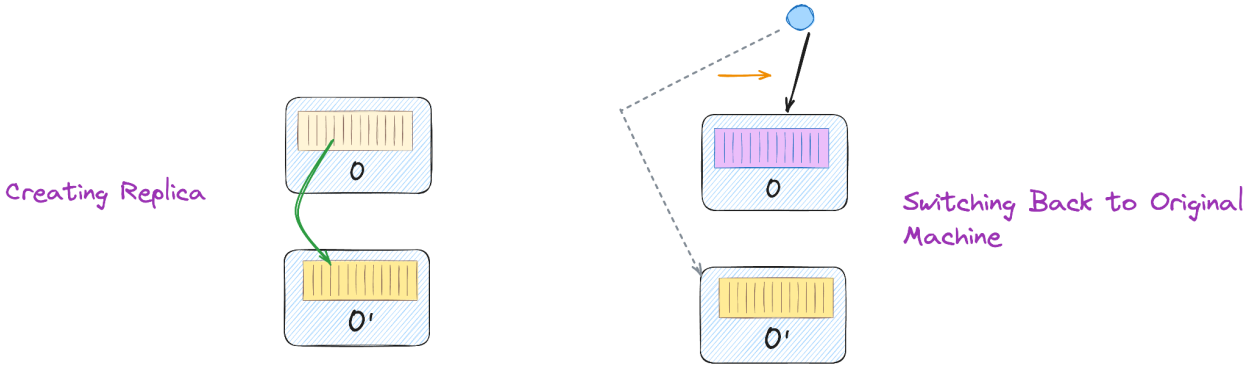


We can stop traffic to the original machines and redirect all traffic to our replicas. Then, we can perform our rehashing and start serving from the original machines again.



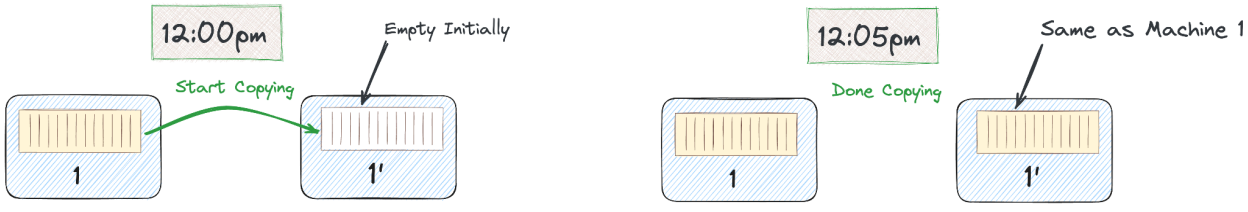
Now, this might seem simple at first, but there are several complications we will encounter. Replication is not a simple process, especially when dealing with live writes. Handling live writes is a big challenge, and here's why:

- **Live Write Challenge #1: Creating the Replica**
- **Live Write Challenge #2: Switching Back to Live Writes on the Original Replica**



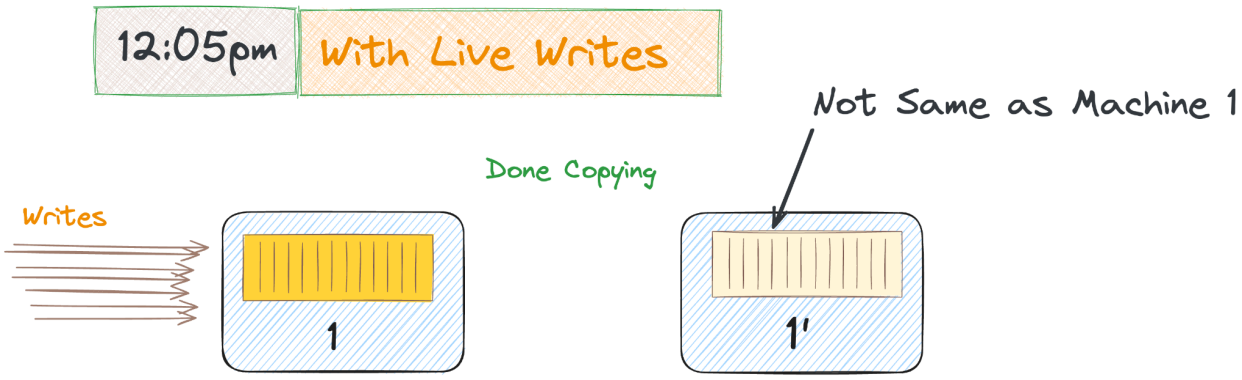
Let's say we have an active cache running. We'll stick to one machine for now. We want to create a replica, which is an exact copy. An exact copy means it's being updated live, at the same time. How can we achieve this?

We can copy all items from machine 1 to machine 1'. However, let's say we start copying machine 1 at 12:00 PM and finish at 12:05 PM. It takes 5 minutes to copy 6GB of data. This is just an estimate; it could be faster or slower depending on factors like network speed, disk I/O, and server load.



But here's the tricky part: while we're copying, the original machine is still receiving and serving requests. This means that during those 5 minutes, new writes are happening on

machine 1 that aren't yet on machine 1'. So, by the time we finish the initial copy, machine 1' is already out of date.

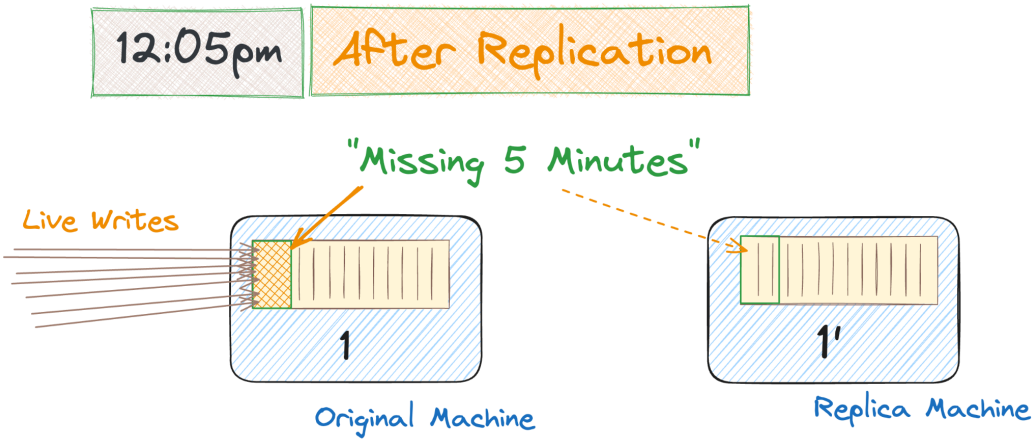


It's like trying to copy a book while someone is still writing new chapters—keeping up with the changes is quite the challenge!

Handling the Delay in Replication

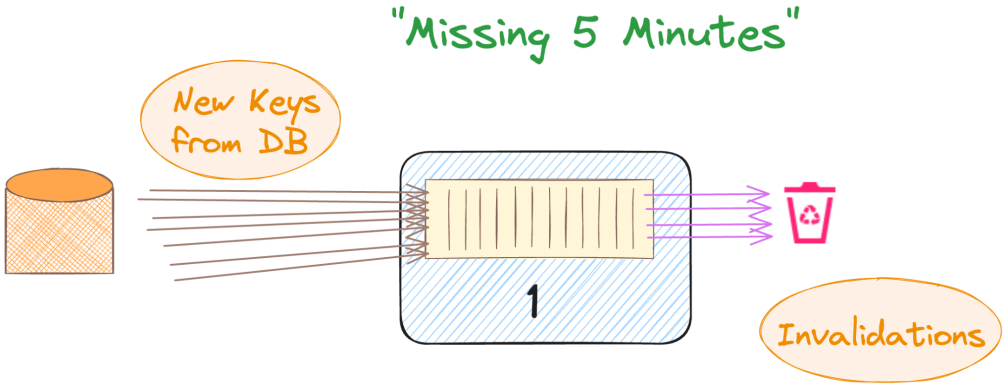
In those 5 minutes while we were copying data, new cache writes were happening. So, our replica is 5 minutes behind.

What do we do? Well, the good news is that this isn't a database. We don't rely on the cache to be the ultimate authority. Even if we lose some writes of new keys, it's fine. The web server will go to the database and cache it again. Some requests will experience a performance hit, but that's probably okay. We need to analyze the effect of these "missing 5 minutes."

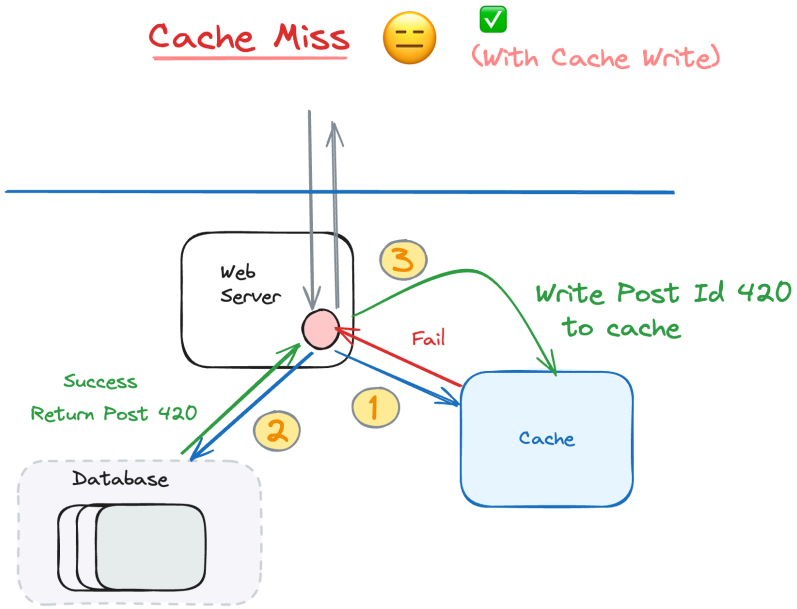


There are two types of writes to a cache like this:
- **New Keys**
- **Invalidations of old keys because the data has changed**

New Keys: When there is a cache miss, the web server puts the key in the cache. This way, future reads hit the cache. In the case of our replica, it's okay if we miss out on 5 minutes of new keys. These will be treated like cache misses and put back into the cache. No problem here, just a minor performance hit.

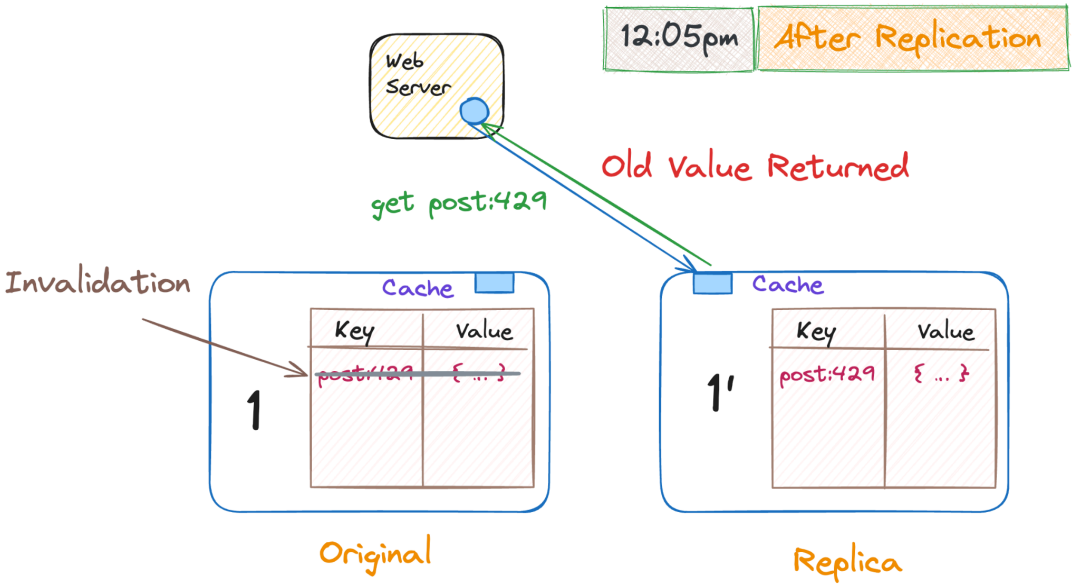


So, while our replica might be slightly out of date, it won't cause major issues. The web server will simply fetch the missing keys again and store them in the cache. Sure, some requests might be a tad slower, but it's a small price to pay for not having to halt everything for the upgrade.



Invalidations of Old Keys: In our cache, when the server needs to update a value, it updates the value in the database and then invalidates the cache entry. This means the old cache entry is removed, and the next time the data is requested, it is fetched fresh from the database and stored in the cache again.

This process can cause an issue during our "missing 5 minutes." Can you guess what it is? Let's look at the impact of our missing data. What exactly are we missing? We're missing cache invalidations. For example, let's say a user edits a post. Normally, the cache entry for that post would get invalidated.

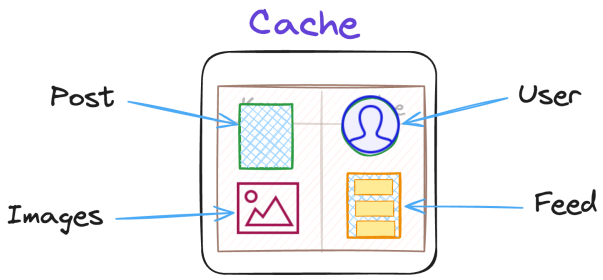


However, during those missing 5 minutes, any invalidations are not copied to the replica. So, if our replica takes over, the cache entry for post 56, for instance, will still hold outdated data. This will continue until the cache entry expires naturally or the user updates it again. This is usually not what we want because it means serving stale data.

So, while missing new keys in our replica isn't a big problem, missing invalidations is an issue.

How can we fix this? In some cases, missing invalidations might be acceptable if the cache entries have a short expiry time, like less than 5 minutes. If it is ok to show stale data for 5 minutes, it might be fine.

For example, let's say we have entries for a user post, and it's okay if edits to those posts take 5-10 minutes to reflect. In this case, the delay isn't a problem. However, remember that a single cache holds all kinds of data. It's one giant table with everything stuffed inside.



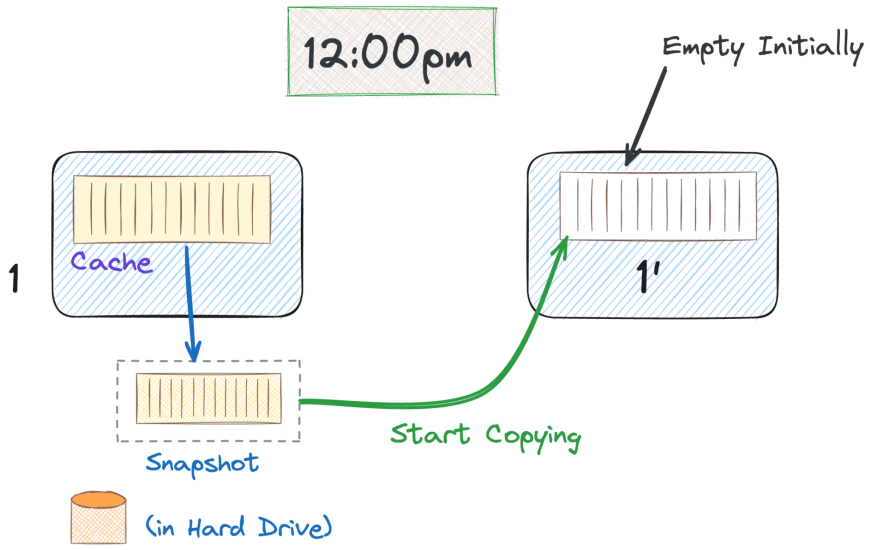
So, we can't just ignore the issue. While delaying updates to posts might be acceptable, it might not be okay for user information or other critical data.

Alright, we can't ignore the issue. We still need to handle invalidations effectively.

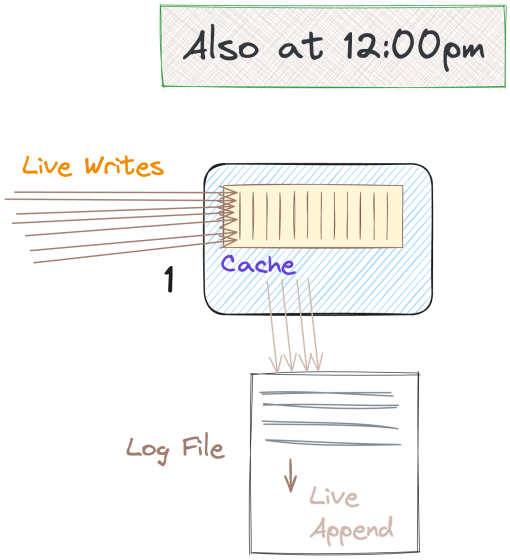
Handling Delayed Cache Invalidations

The solution is quite simple and it's a technique we often use in system design.

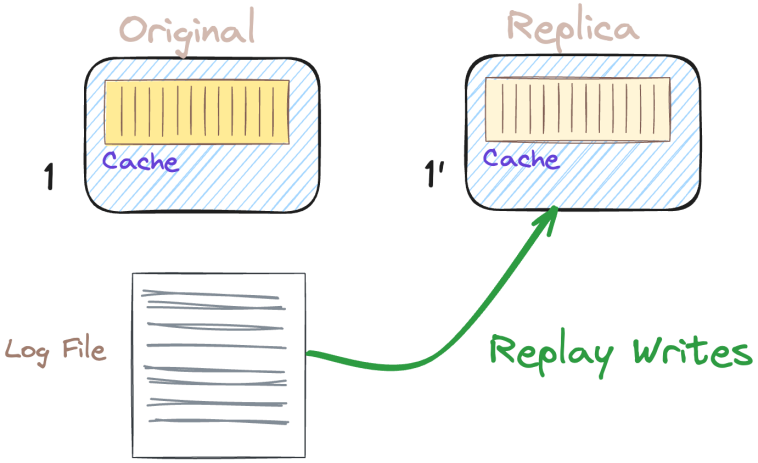
While the replication is happening, we keep a log of writes in our original cache. Let's say we start copying at 12:00 PM. We take a snapshot of our hash table at 12:00 PM and start transmitting it to our replica.



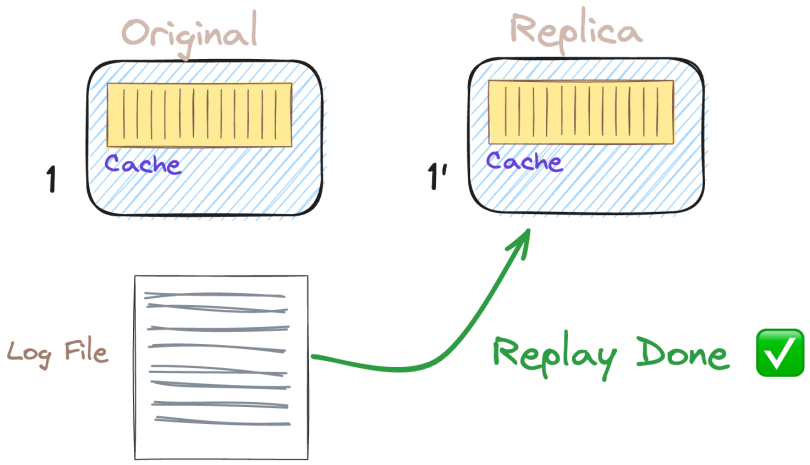
At the same time, we begin recording all the writes (new keys and invalidations) in a log. This log lives on the original machine and is simply a file we keep appending to.



At 12:05, we've finished copying the 12:00 PM snapshot to the replica. Now, we start replaying the recorded log's commands into the replica, one by one. This way, the replica catches up with any changes that occurred during the initial copying process.



At some point, we will catch up, and the replica will be up to date.



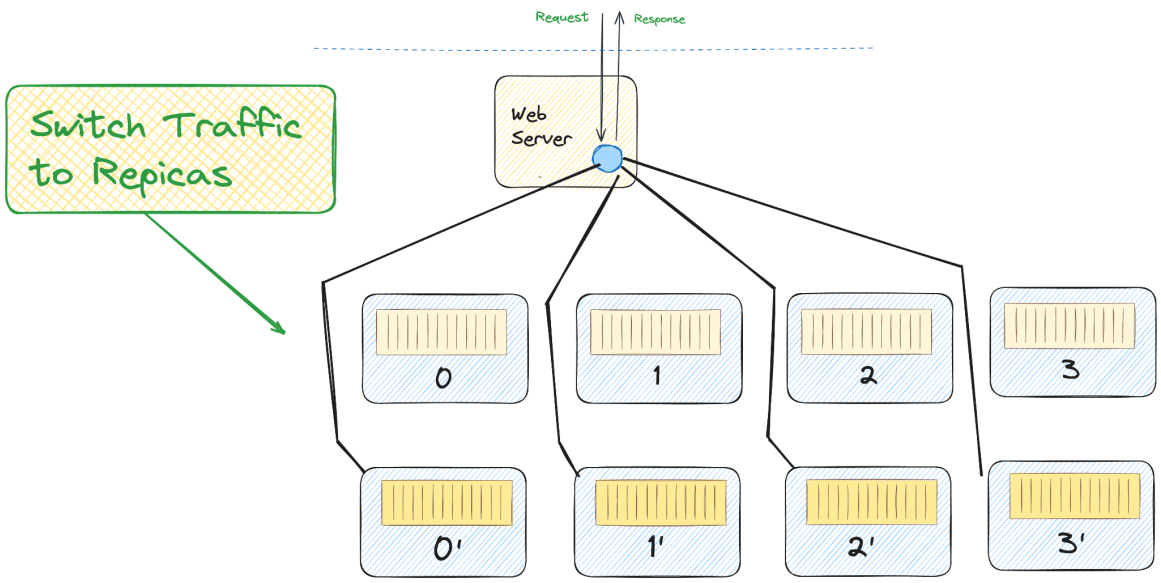
This approach ensures that the replica is updated with all the writes that happened during the replication process, maintaining consistency between the original cache and the replica.

Now, this log file is actually a live file. It is constantly being appended to with new writes and invalidations. We can use this file to keep the replica up to date continuously.

But let's set aside the idea of keeping the replica up to date for now. For now, we only care about a one-time update before we start the remapping process.

Once the file is done, and the replica is caught up, we can transfer requests to our replica. Now, our original machine is free, and we can remap it.

Let’s put this in the context of our 4-machine setup. We replicate all 4 machines and start serving requests from the replicas.



This process frees up the original machines for remapping.

However, it's important to note that this is quite a tedious process, especially for a system that is serving live requests 24/7. This continuous log and update method helps ensure data consistency, but it requires a lot of coordination.

We will face the exact same problem when we switch back to the remapped cache servers. We'll need to replay the logs and ensure the machines are up to date before they can start serving requests again.

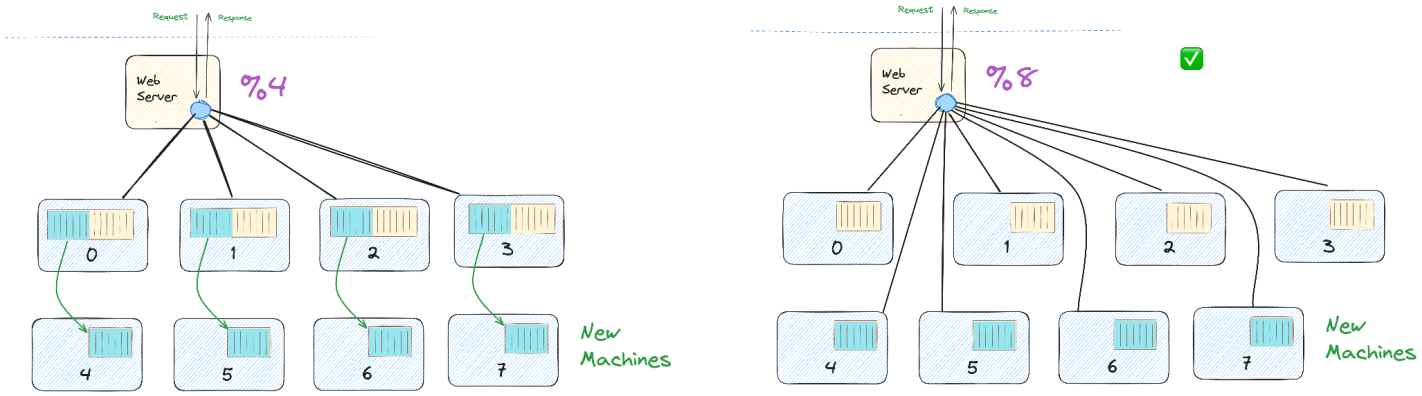
Let's do a recap. We discovered that to add a new machine to our 4-machine setup, we need to remap all the keys. This process will cause downtime. To prevent that, we can use replicas as an interim cache.

The downside of using replicas is that we'll need to rent out the same number of machines as our cache currently has, and it's a tedious process. However, it’s doable.

This brings us to another solution to our problem: preventing the need to remap keys by doubling the number of machines.

Can we Double the number of Machines?

Doubling Machines: This is a clever trick. Let's revisit our original scenario. We are running 4 machines and want to increase our capacity. So, we buy 4 more machines. With these additional machines, we can take half the keys from each existing machine and move them to the new ones.



To make this work, we adjust the hash function to distribute the keys across the new setup. Instead of using hash % 4, we switch to hash % 8.

How Doubling Machines Works: The Math

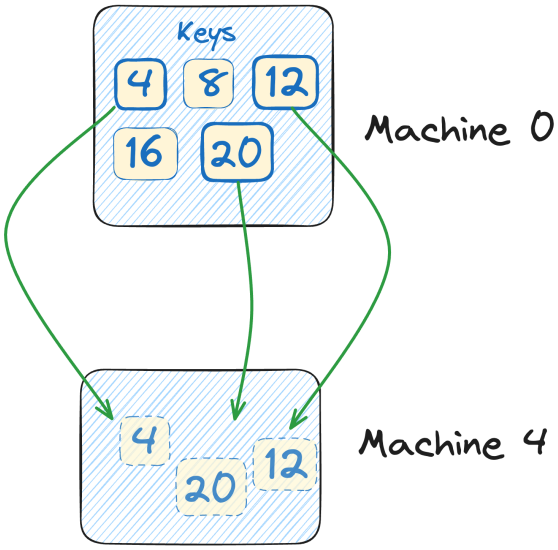
The math behind doubling machines works out well. Here's how it goes:

<diagram>

Take Machine 0. It has keys like 4, 8, 12, 16, 20, and 24. If we mod these by 8, we get:

- `4 % 8 = 4`
- `8 % 8 = 0`
- `12 % 8 = 4`
- `16 % 8 = 0`
- `20 % 8 = 4`
- `24 % 8 = 0`

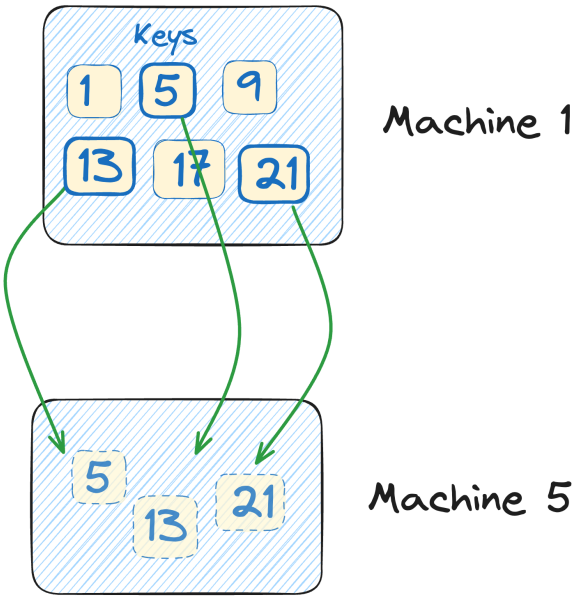
These results mean keys that used to be in Machine 0 will now be in either Machine 0 or Machine 4.



Now, let's look at Machine 1 with keys 1, 5, 9, 13, 17, and 21. If we mod these by 8, we get:

```
-  1 % 8 = 1
-  5 % 8 = 5
-  9 % 8 = 1
- 13 % 8 = 5
- 17 % 8 = 1
- 21 % 8 = 5
```

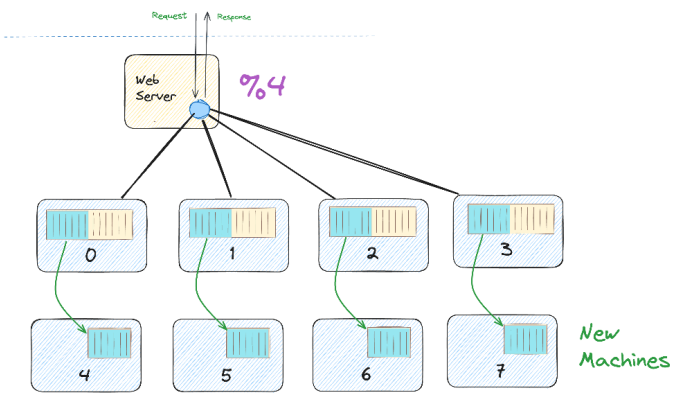
These results mean keys that used to be in Machine 1 will now be in either Machine 1 or Machine 5.

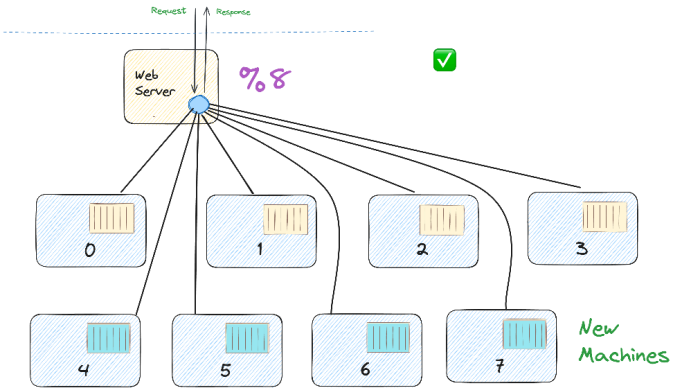


So, by doubling the number of machines and changing our hash function from hash % 4 to hash % 8, we can evenly distribute the keys across all 8 machines.

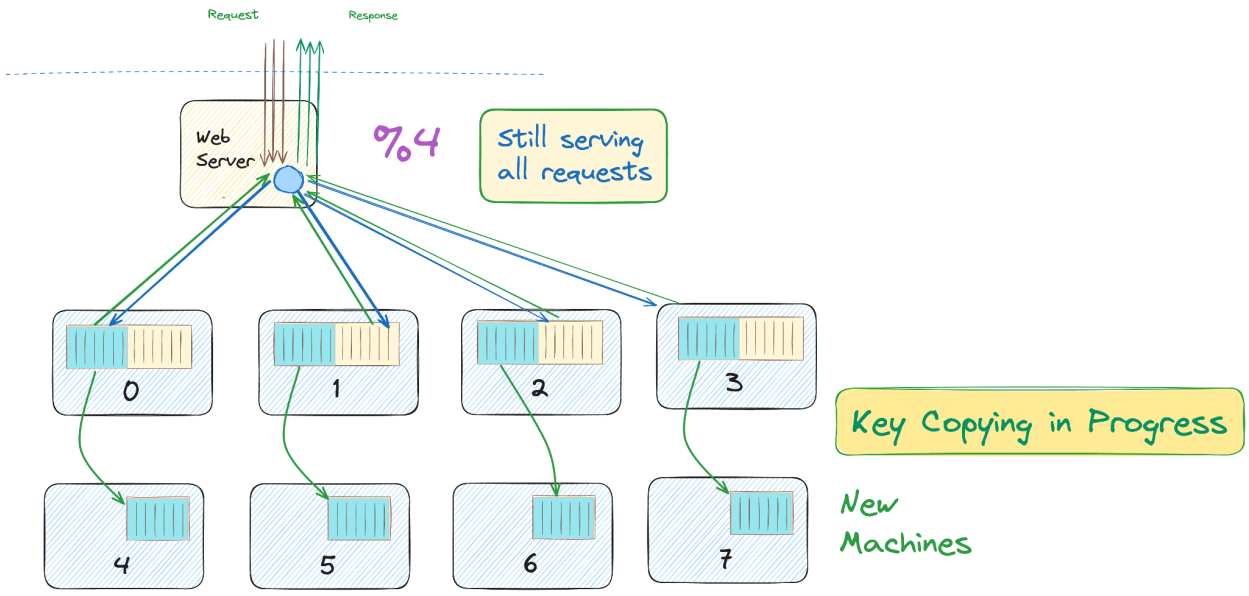
Here's how we do it:
1. We move half the keys from each existing machine to a new machine.
2. We update modulo function to hash % 8

And we're in business.





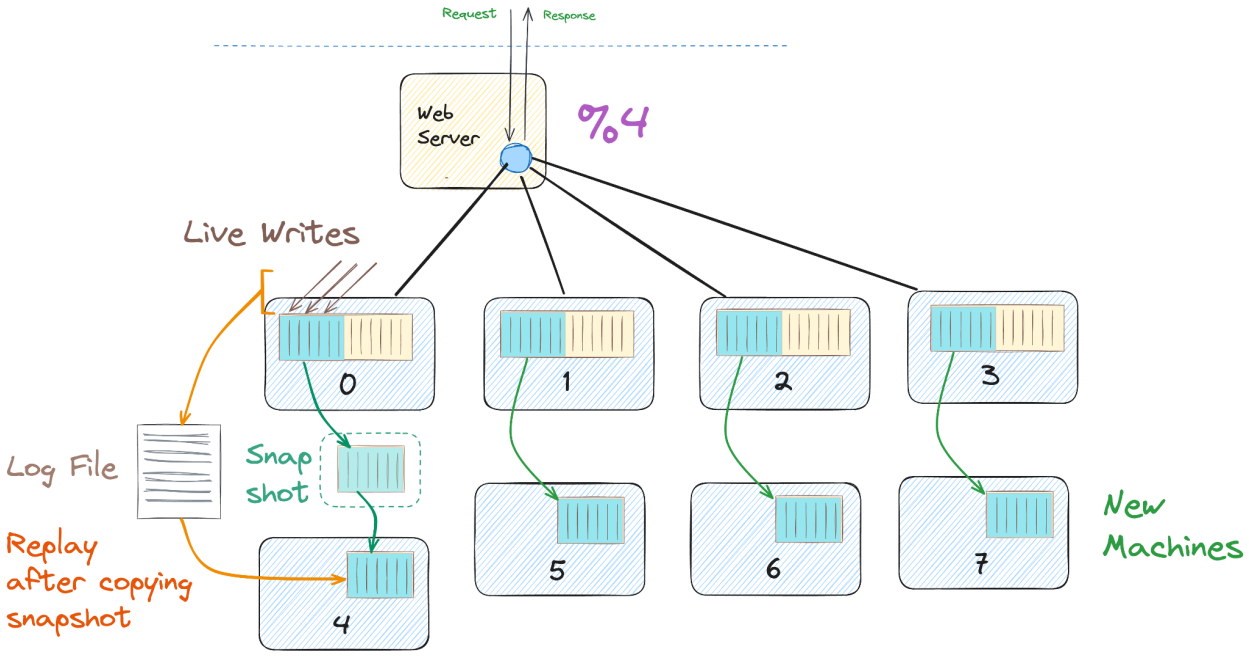
What's the benefit here? We can keep serving requests from the original 4 machines while this process is happening—no need for downtime.



Steps for Doubling Machines

Once we start using our new setup with hash % 8, we can delete the keys we moved to the new machines. We don't need extra machines for backup anymore.

Here are the steps again:

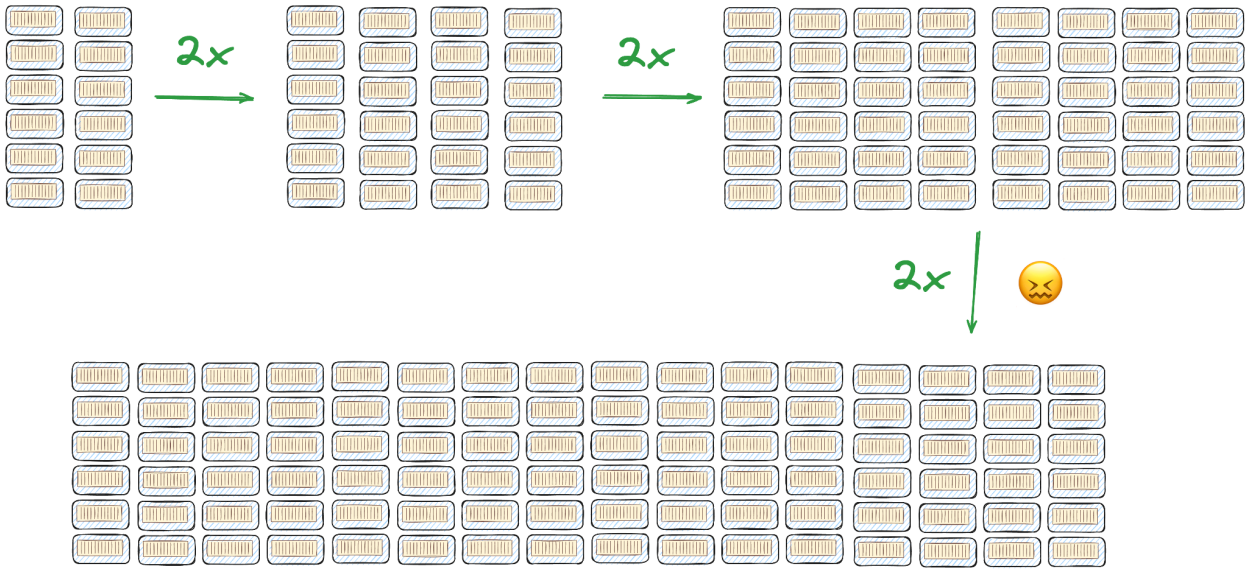


1. **Copy Half the Keys**: Move half the keys from each original machine to the new machines.
2. **Keep a Replica Log**: Record all live requests while copying to make sure no data is lost.
3. **Replay the Log**: After copying, replay the log entries on the new machines to update them with any changes.
4. **Update the Modulo Function**: Change the hash function from hash % 4 to hash % 8 to spread keys across 8 machines.
5. **Verify Requests**: Make sure requests are being correctly handled by the new setup.
6. **Delete Old Keys**: Remove the keys that were moved from the original machines.

And voila! We have a working memcached setup that doubles in size each time.

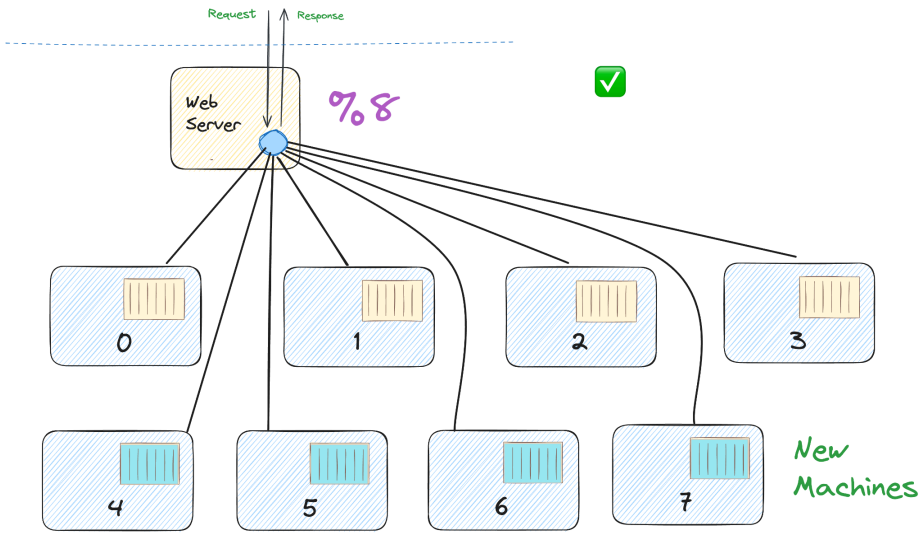
Downsides of Doubling Machines

However, there is a downside to this approach—we need to double the number of machines every time: 2, 4, 8, 16, 32, 64. This quickly becomes tedious and impractical.



Imagine adding 32 machines at a time. It seems unnecessary and overly complex. Eventually, we'll need to add over 100 machines at a time! Ideally, we want the flexibility to add any number of machines we want, but this method doesn't provide that freedom.

But hey, it works for now. Congrats! We've scaled our system to 8 machines. Our site is thriving, and so is our cache performance!



Ch 5. Scaling to 10 machines - Consistent Hashing

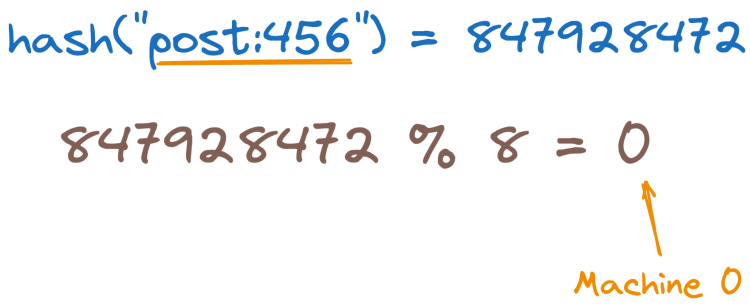
A better way to scale our servers is through consistent hashing. If you are familiar with the concept, feel free to skip this section.

Consistent hashing is a common method used to scale databases, and it's also used by Facebook's memcached to scale to thousands of servers.

Let's say we have 10 machines. We want to distribute our key space across these 10 machines.

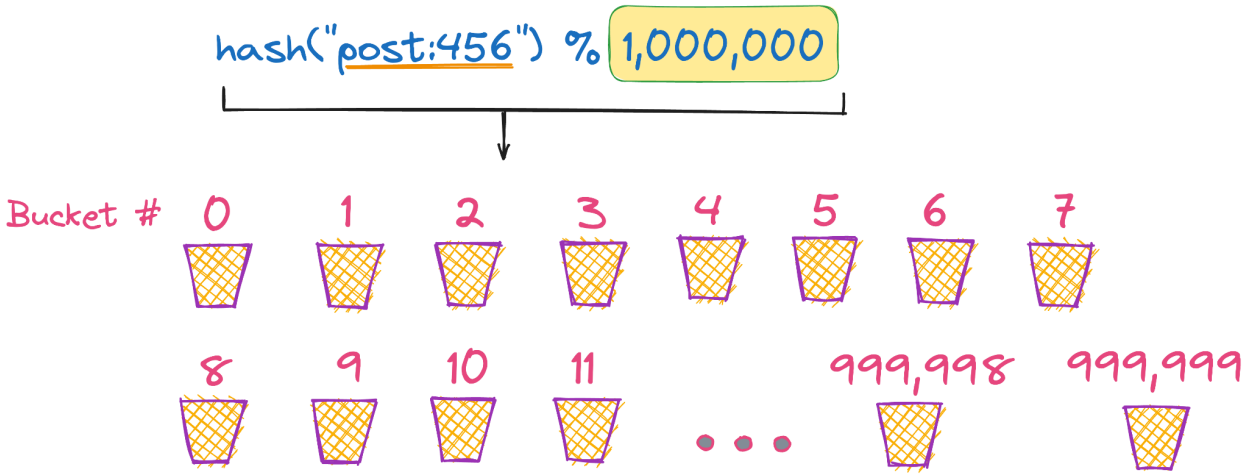
How does the Key Space work in Consistent Hashing?

What is our key space? Our key space is the hash(key) we used earlier to find machine numbers. For example, when we look for 'post:456', we do hash('post:456') % 8. This means we take the hash value of 'post:456' and divide it by 8, using the remainder to determine which machine stores the key. The hash(xyz) function returns a large number, and the % 8 operation splits it into 8 parts.

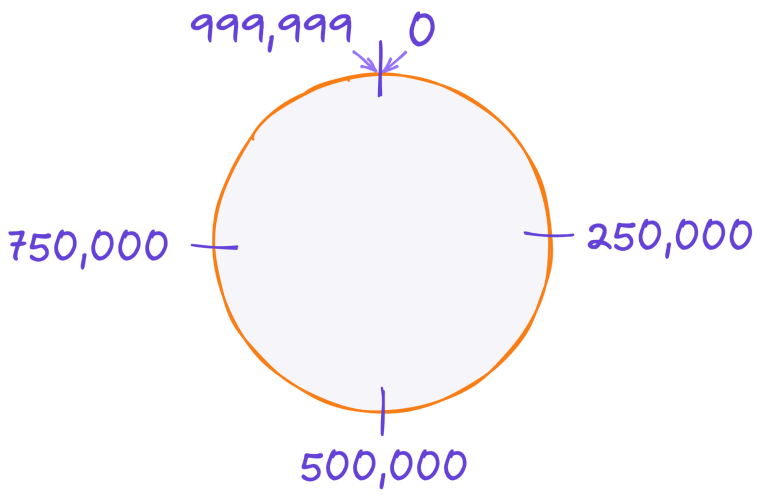


With consistent hashing, we divide the key space into many small parts, called buckets. This way, even if we scale to 10,000 machines, each machine handles hundreds of buckets.

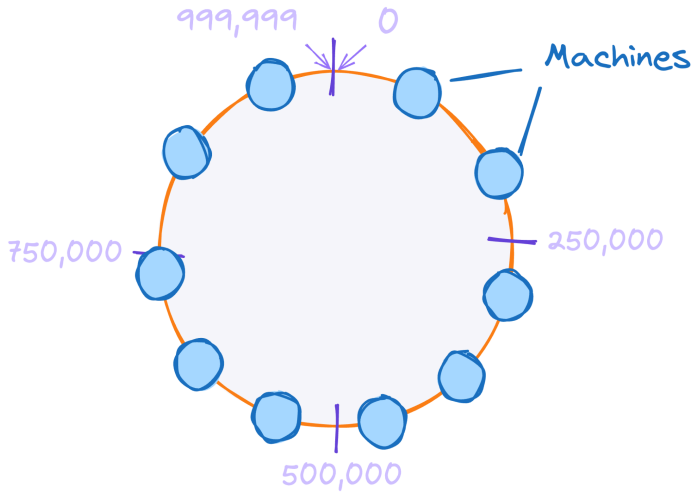
Why use small buckets? Because they are easier to move around when machines are added or removed. Let's say we divide our key space into 1 million parts using hash(key) % 1,000,000. This gives us buckets numbered from 0 to 999,999.



Now, here's the magic. We place these buckets on a ring, like a clock face that starts at 0, goes up to 999,999, and then wraps around back to 0.



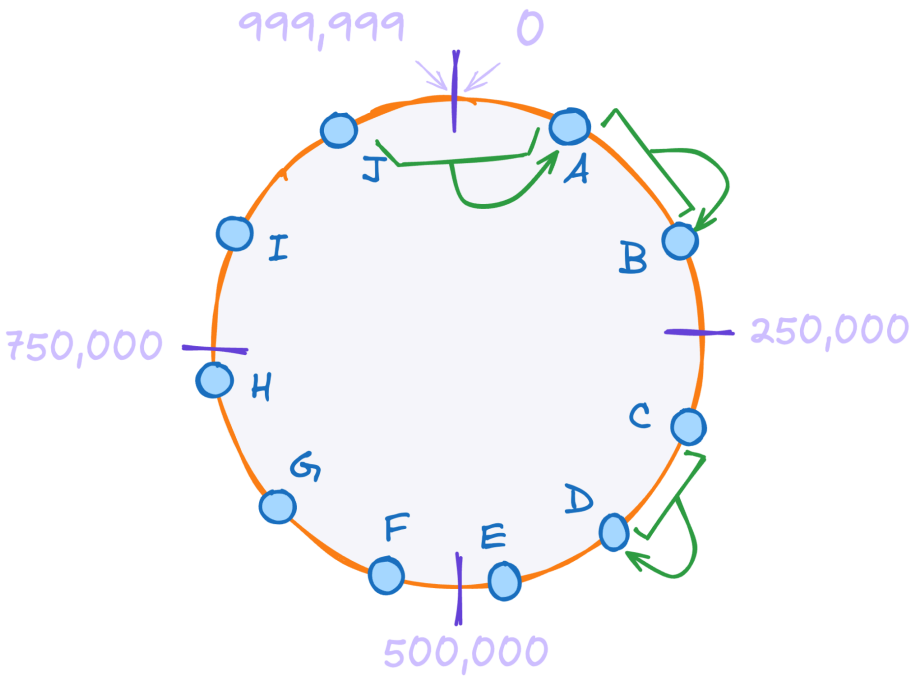
Let's say we have 10 machines. We place each machine at different points around the ring. Ideally, these machines are evenly spread out. If not, we can adjust to make sure each machine has about the same number of buckets.



Look closely at the ring. We give each machine the responsibility for the part of the ring next to it.

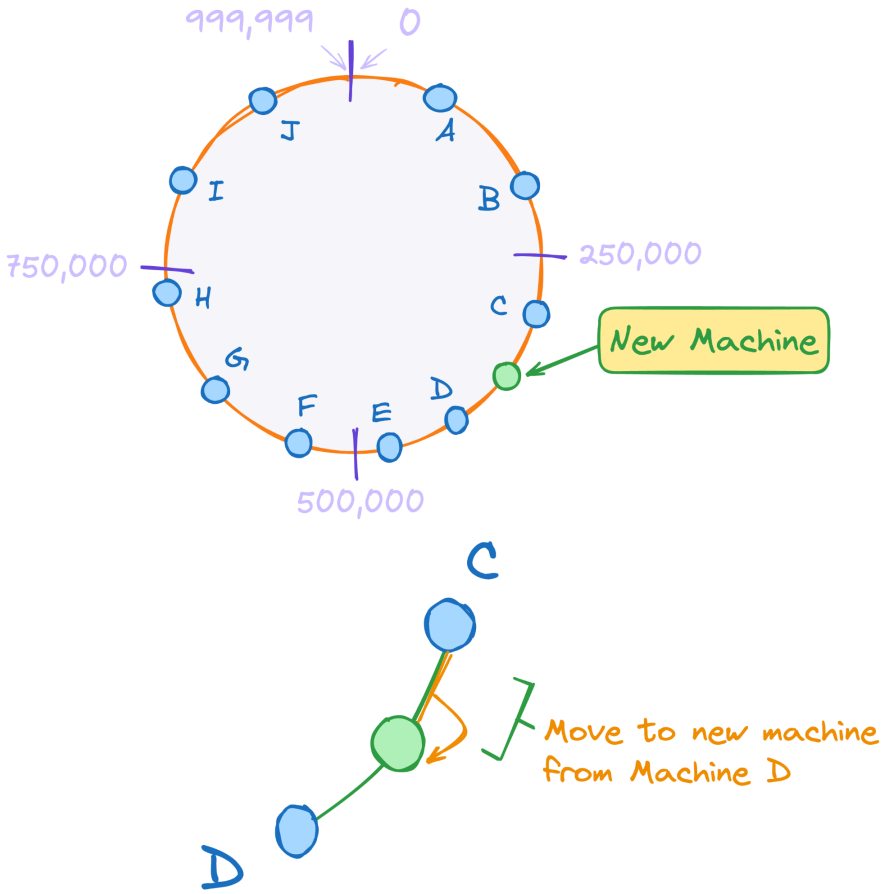
For example,
Machine B handles buckets 56,109 to 181,472.
Machine C takes care of buckets 181,473 to 321,388 and so on.
Machine A covers the end of the range, from 941,375 to 999,999,

and wraps around to handle 0 to 56,108.



What's the benefit of this? When we had just 8 buckets, adding a new machine was hard. We had to change the formula from % 8 to % 9, which meant remapping every key to new buckets. All the keys had to be remapped because the number of buckets changed. Even with our doubling solution, half the keys needed to be moved at once.

Now, with consistent hashing, we simply add the new machine to the ring. This new machine takes over some of the buckets from an existing machine. The existing machine then moves those buckets to the new machine, and we're done.



Let's recap what we've covered:

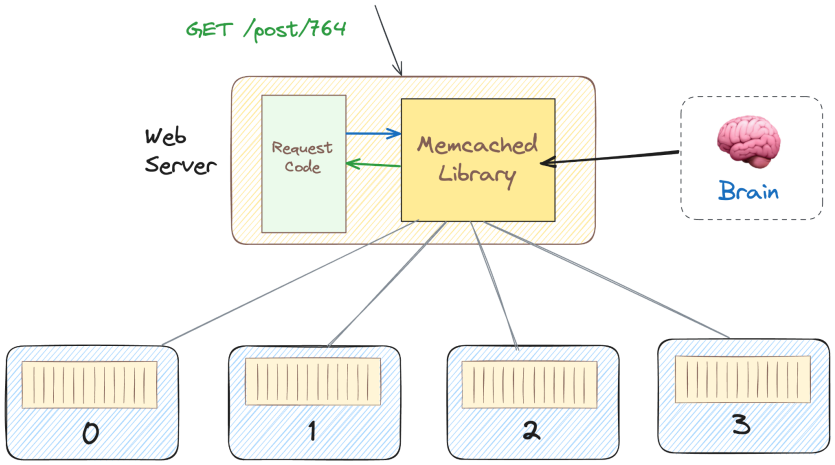
- **Modulo-Based Partitioning**: We started with this, but found we had to remap all the keys every time we added a machine.
- **Making It Work**: We explored two ways to improve this:
- **Replicas**: Creating a replica of the entire cache while remapping happens.
- **Doubling Machines**: Doubling the number of machines each time. This method has drawbacks because it requires large upgrades every time.
- **Consistent Hashing**: We then discussed consistent hashing, which is more scalable and efficient.

Should we have Masterless Nodes in Consistent Hashing?

Consistent hashing has some extra benefits we won't get into. One big benefit is it can work without a central coordinator.

In our current model, the web server's client library figures out which machine to contact. This client library acts as our "coordinator" or "master."

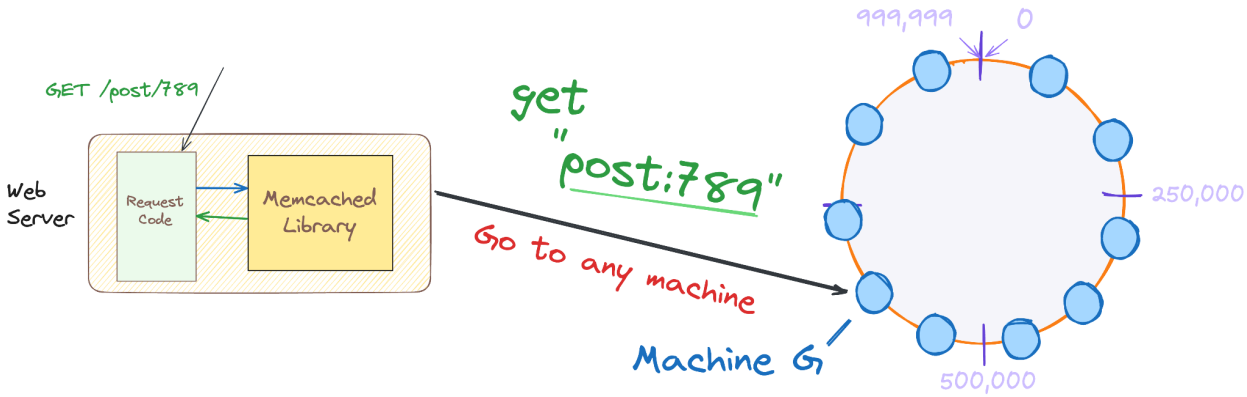
If a new machine is added or if a machine goes down, the client library must handle these changes. It is the brain of the cache, while the individual machines are just simple hash tables.



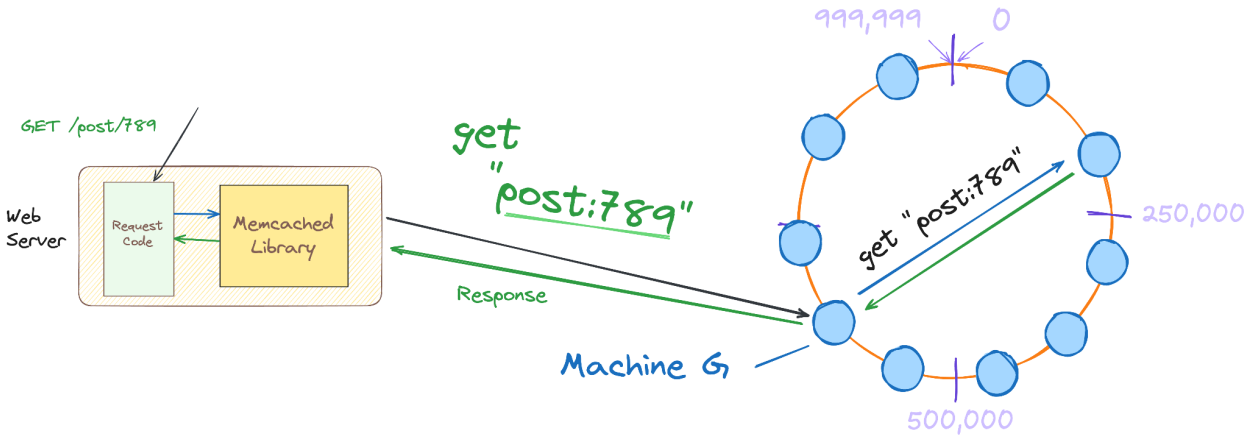
In many databases, consistent hashing doesn't need a master. This means the logic to manage nodes and handle changes is spread out to each node, not just in one central place.

Here's how it works:

The web server wants to fetch the key "post:789". It contacts any machine in our cache and asks for the key "post:789".

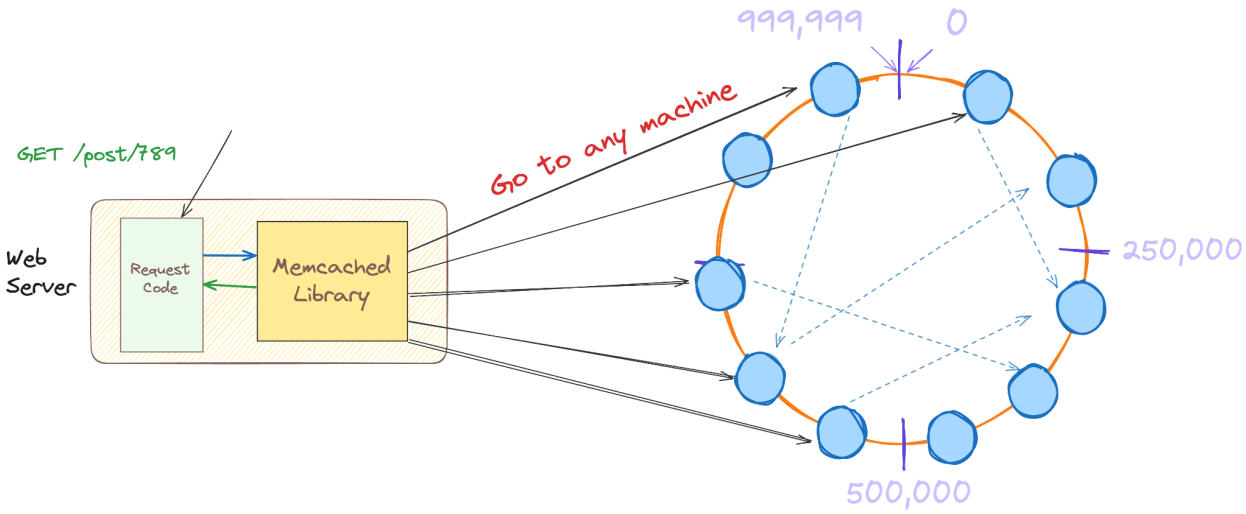


Machine G may or may not have the key. It runs the hash algorithm -> hash(key) % 1,000,000 to find out which machine has the key.



Machine G then fetches the key from the correct machine and returns it to the server.

Do you see the benefit here? The "brain" has been moved from one machine (the web server) to all the cache machines. If there are 1,000 cache machines, each one can handle requests. This increases our possible throughput.



So far, our "brain" has been doing minimal work, just calculating which machine holds the key. In reality, it will do other things like:
- Coordinating when a new machine joins the ring (key transfers)
- Managing replication
- Handling machine failures

So, it makes sense to distribute the work of the brain.

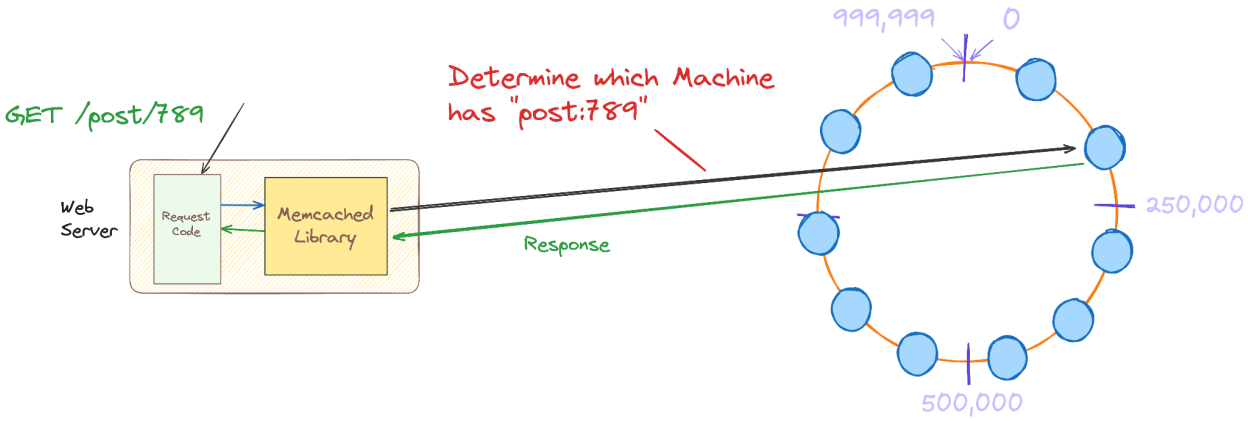
Not Using Masterless Nodes in Consistent Hashing

Now, let's talk about how we'll set up our cache. We're not going to use the masterless approach. While useful, there are reasons why having a master or coordinator makes more sense for memcached at scale.

We'll follow a strategy similar to Facebook's, and later we'll see why this works better for a large, scalable cache.

So for now, we'll use consistent hashing to create buckets and place them in a ring, but the nodes themselves won't have any "brain." They will simply serve as lookup tables.

In our setup with a single web server, our client library will determine which machine to fetch the keys from.



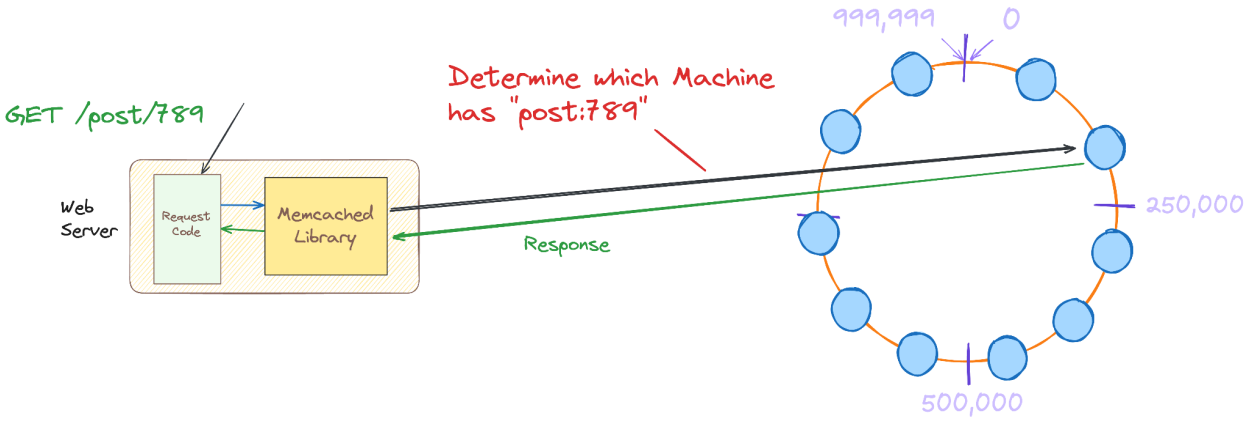
So far, we've ignored the roles of the coordinator, such as replication and failure handling. Let's see how we can handle those next.

Ch 6. Scaling to 100 machines - McRouter

Alright, our current setup worked for a small home setup, but it has some serious drawbacks and missing features. Here are a few issues:

- **Machine Failure**: What happens if a machine fails? How do we handle those requests?
- **Read Replicas**: We want read replicas to manage read load. How can we set this up?
- **Monitoring**: How do we monitor the health and errors of each machine?
- **Smart Handling**: What if we need more intelligence in our system? For example, if we want a separate pool of machines for specific keys, like celebrity posts that get a lot of reads. We might want 4 read replicas for those posts. We'll need some smart handling for this.

In our current setup, the client library has simple functions: it calculates the bucket number and maps it to the correct machine.



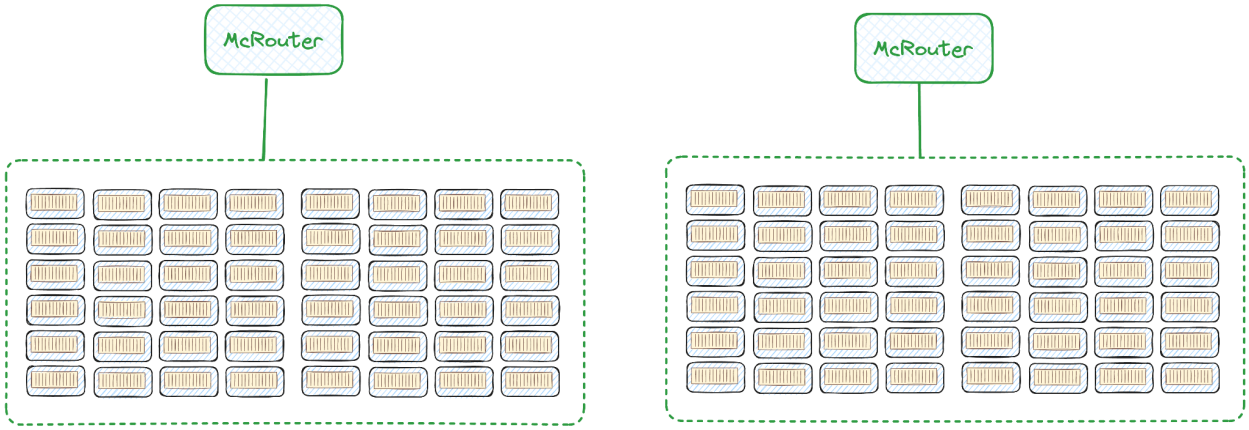
As we need more intelligence, it makes sense to move this "brain" to a different process.

The word "process" might make us think it lives on the same machine as the web server, but that's not always the case. This process might run on a dedicated master machine.

Introducing Mcrouter: The Smart Coordinator

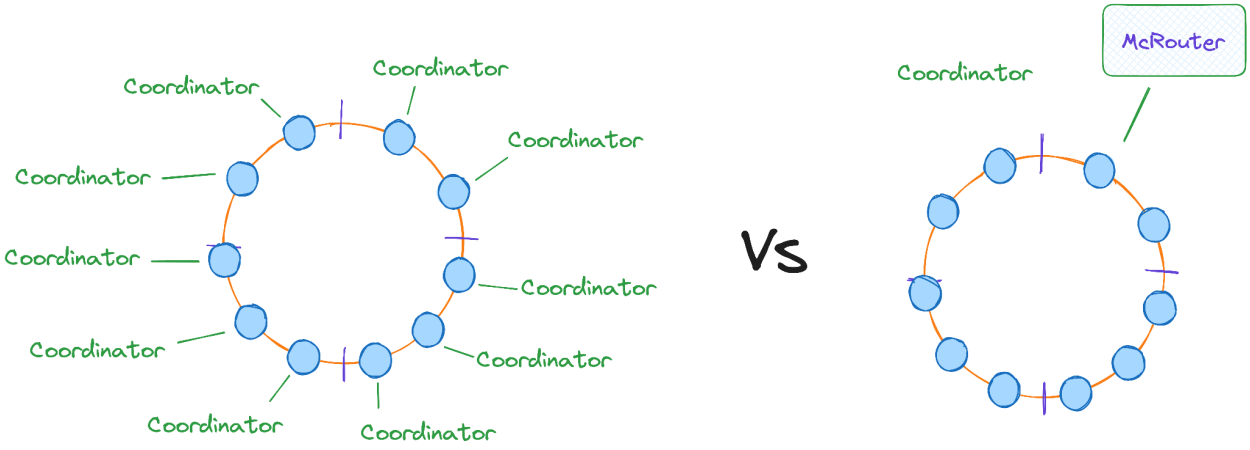
Facebook calls this process 'Mcrouter.' It acts as a smart coordinator or 'master.' Web servers send requests to Mcrouter, which then routes the requests to the appropriate memcached server.

For example, if we have 10,000 machines (as Facebook does), we can have several Mcrouter machines to handle the heavy load. If each Mcrouter can manage 100 memcached machines, we would need 100 Mcrouter instances to handle all the traffic.



Why Use Mcrouter?

The main reason to use Mcrouter is to add a lot of intelligent features to our memcached cluster. With just masterless consistent hashing, it's harder to add complex intelligence to every single node.



For example:

- **Configuration Management**: It's easier to store configurations, like which machines are active and where to send specific requests, in one or a few machines instead of 10,000 machines.
- **Adding New Features**: Adding new features, like failover handling, is simpler to test and deploy on a single machine rather than across 100 machines.

- **Debugging**: Finding and fixing bugs in the coordination logic is easier when it occurs in a single machine or a few machines instead of 10,000.
- **Metrics and Stats Collection**: Gathering metrics and stats for the whole cluster is more practical from a single location rather than every machine. This makes monitoring and observability more efficient.

As we can see, a masterless approach works for simple intelligence and fewer machines. But as soon as things get complex, decentralizing everything doesn't make sense.

(Lesson: Decentralizing everything doesn't make sense, even in a "distributed" system.)

Building Our Cache with Mcrouter

Now we're getting into serious business. So far, we've covered scaling a cache to handle millions of users. But now, we're talking about handling billions of requests per second with thousands of machines—industrial-scale stuff.

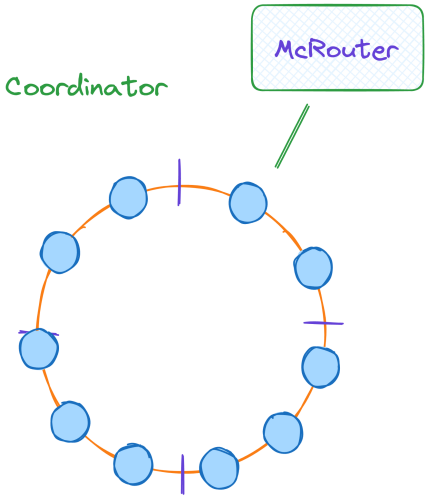
For your interview, you should understand both basic and extreme scaling. Prematurely optimizing for extreme scale can make you seem like you're just repeating a research paper you've memorized. It's important to understand the journey from a single machine to large-scale systems. You now know the details of scaling from scratch and the major issues you encounter along the way. You also know different approaches and why they do or don't work, allowing you to evaluate strategies effectively.

Notice that many strategies we tried didn't work well. Going through this journey is crucial. You need to practice trade-offs and system evaluations. We've done that here.

In an interview, you won't need to go into as much detail as we did in this book.

We are very detailed here to explain everything thoroughly. You don't need to be this detailed in an interview.

Ok, so let's get started. We now have a Mcrouter process running on a different machine as follows:

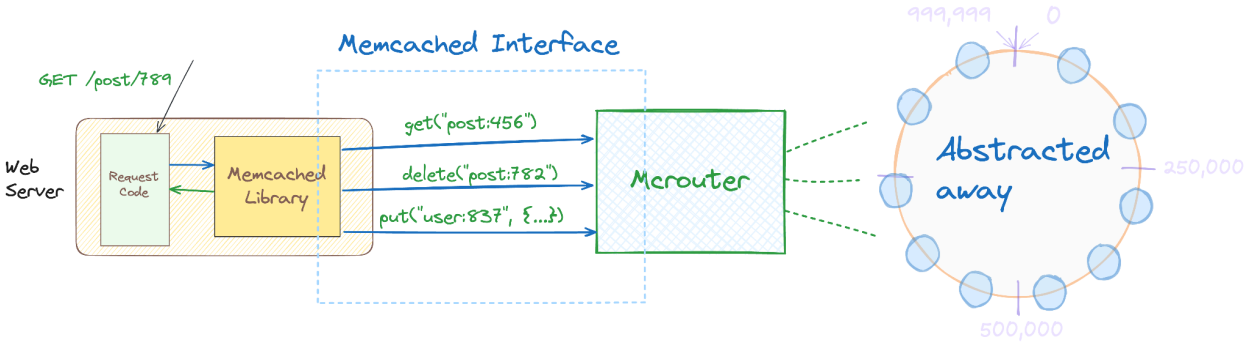


We make two assumptions:

1. **Mcrouter is running on its own machine**: This is for simplicity. In reality, it can be a process running among other processes. For example, at Pinterest, you have a Mcrouter process running on every web server machine.
2. **We only have one instance of Mcrouter running for now**: We'll add more later. For now, let's say we have 1 Mcrouter machine and up to 100 memcached servers running. Wow, that's some scale!

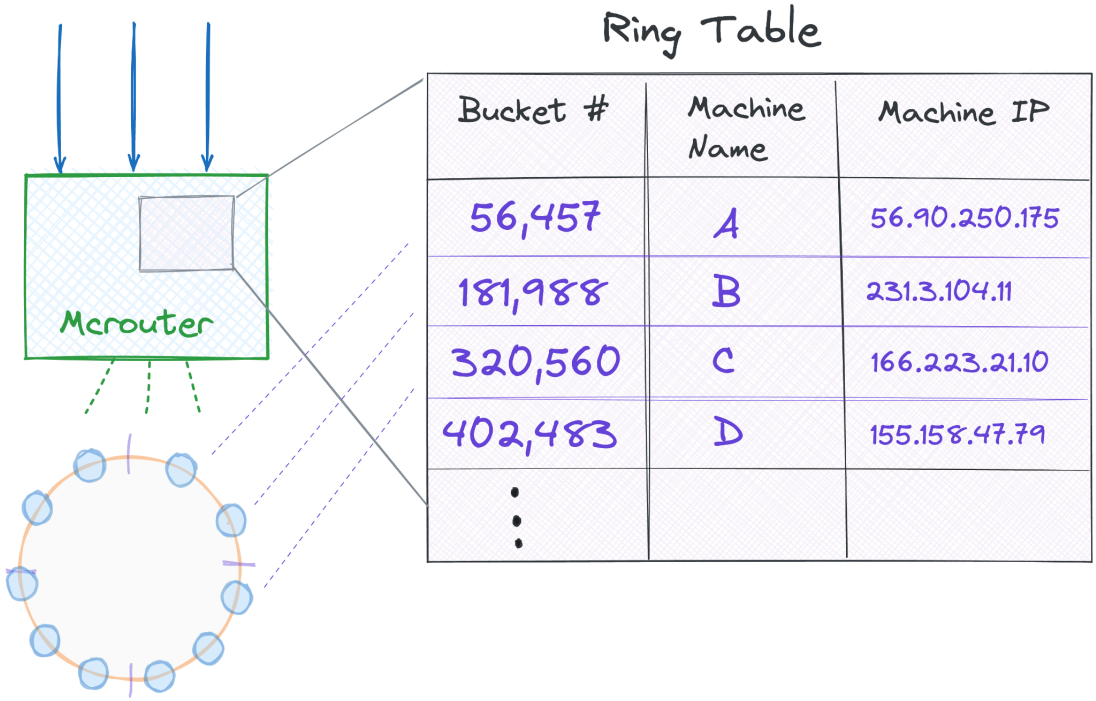
How do we build this process? Let's start with the most simple feature—routing requests using consistent hashing.

When a request comes in, we need to find the machine responsible for it on the ring. Our cache can handle three types of requests: get(key), put(key), and delete(key). This is the interface. To a web server, it looks like it is interacting with a single memcached machine. The web server doesn't know anything about Mcrouter. In our code, we treat Mcrouter as if it were a memcached instance, just like we did in the early days.



How does Mcrouter work?

Within Mcrouter, there is a table that contains each machine's position on the ring.

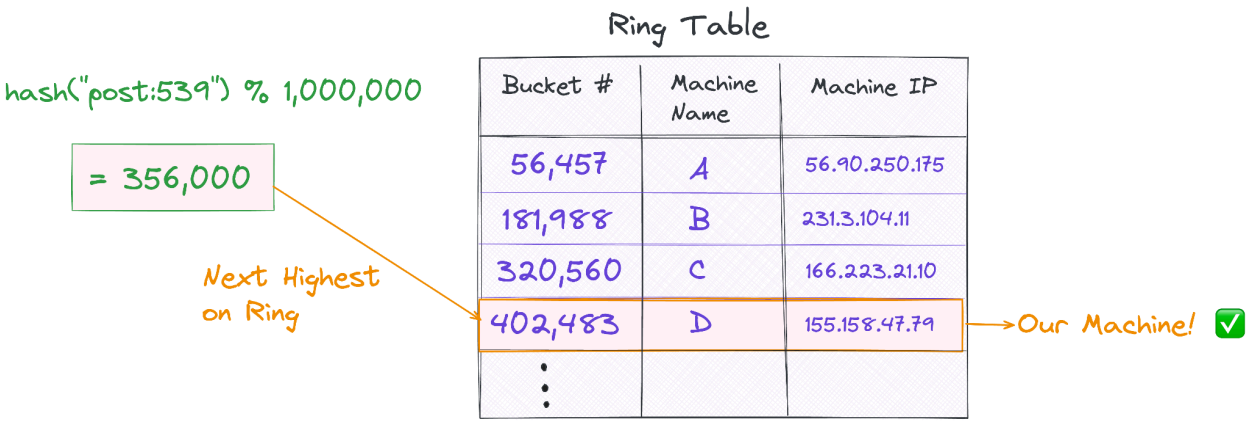


Finding the Machine for a Key

Given a key, how do we find its machine on the ring? Think algorithmically. The answer is: we use binary search! Let's break it down:

1. **Compute the Hash**: Say we are looking for the key "post:345". We compute the hash and get hash("post:345") = 125,101. This is our bucket number.

2. **Find the Next Highest Value**: We need to find the machine responsible for this bucket. We do this by finding the next highest value in our table.



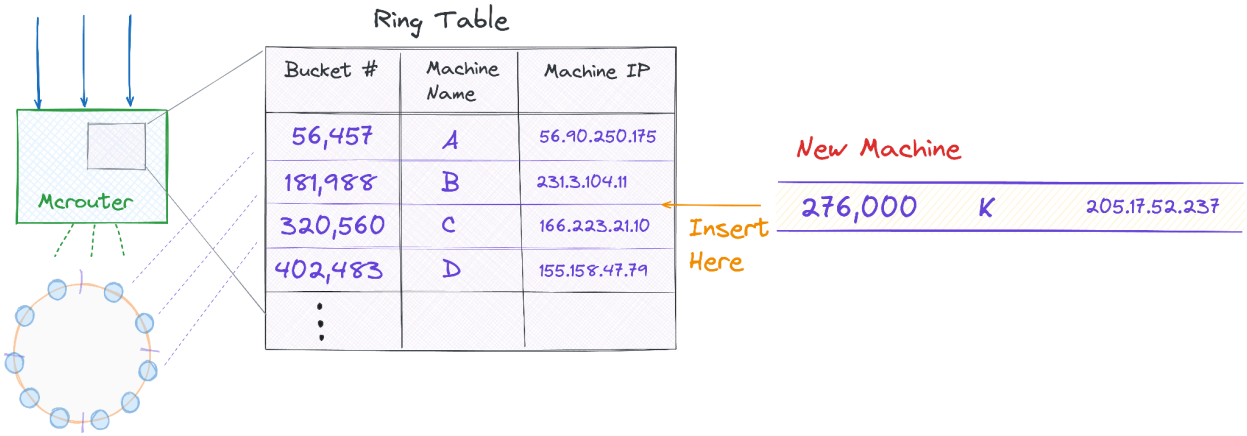
We perform a binary search through the keys. Binary search is efficient, taking log(N) time, where N is the number of machines. This is fast enough for our needs.

If we want to be even faster and have a fixed number of buckets (like 1 million), we can create an array with 1 million entries and populate it with the corresponding machines. This way, finding the machine becomes a matter of a simple array lookup, which is extremely fast.

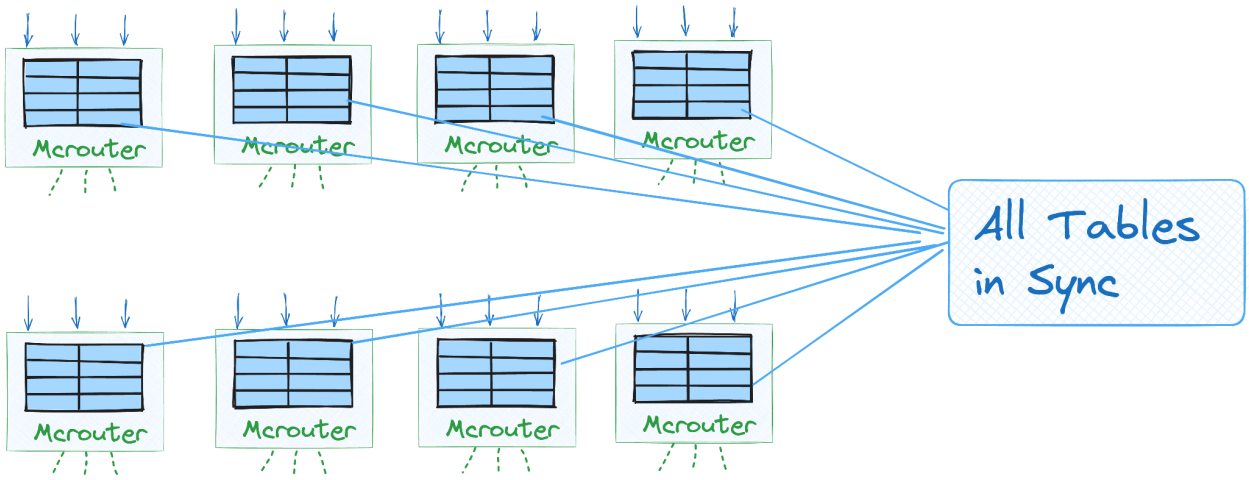
If we were to use an array for optimization, storing 1 million integers would take 32MB (1,000,000 * 32 bytes). While this is doable, it's likely overkill.

Managing the Consistent Hash Table

Let's talk about the consistent hash table in our Mcrouter. This table is crucial because it controls how requests are routed. When we add a new machine, this table needs to be updated to include it.

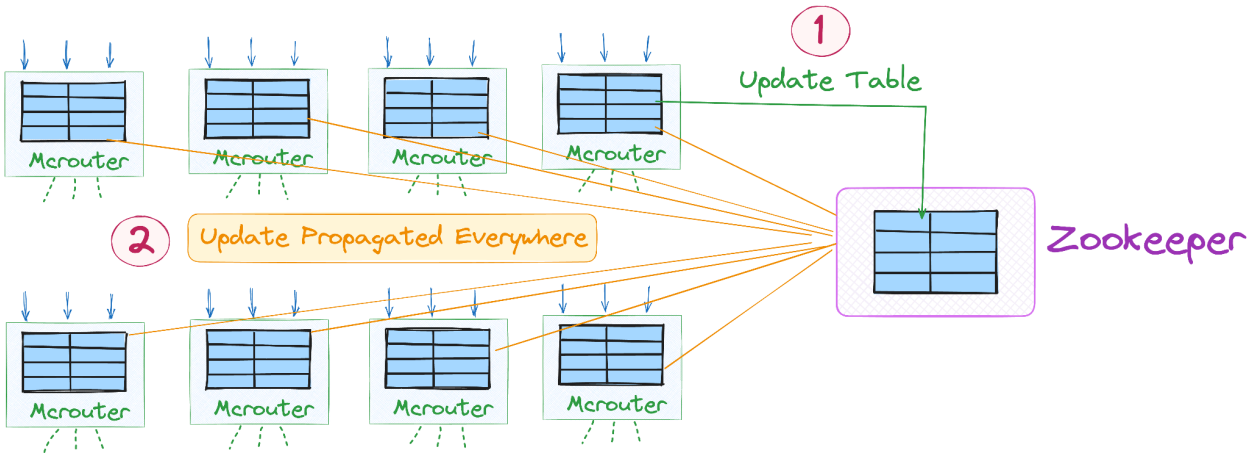


In one Mcrouter instance, updating the table is simple. But when we have 100 Mcrouter instances, we need to update each one.



This table is a very important configuration. To keep it in sync across all machines, we use Zookeeper, which is designed for this purpose.

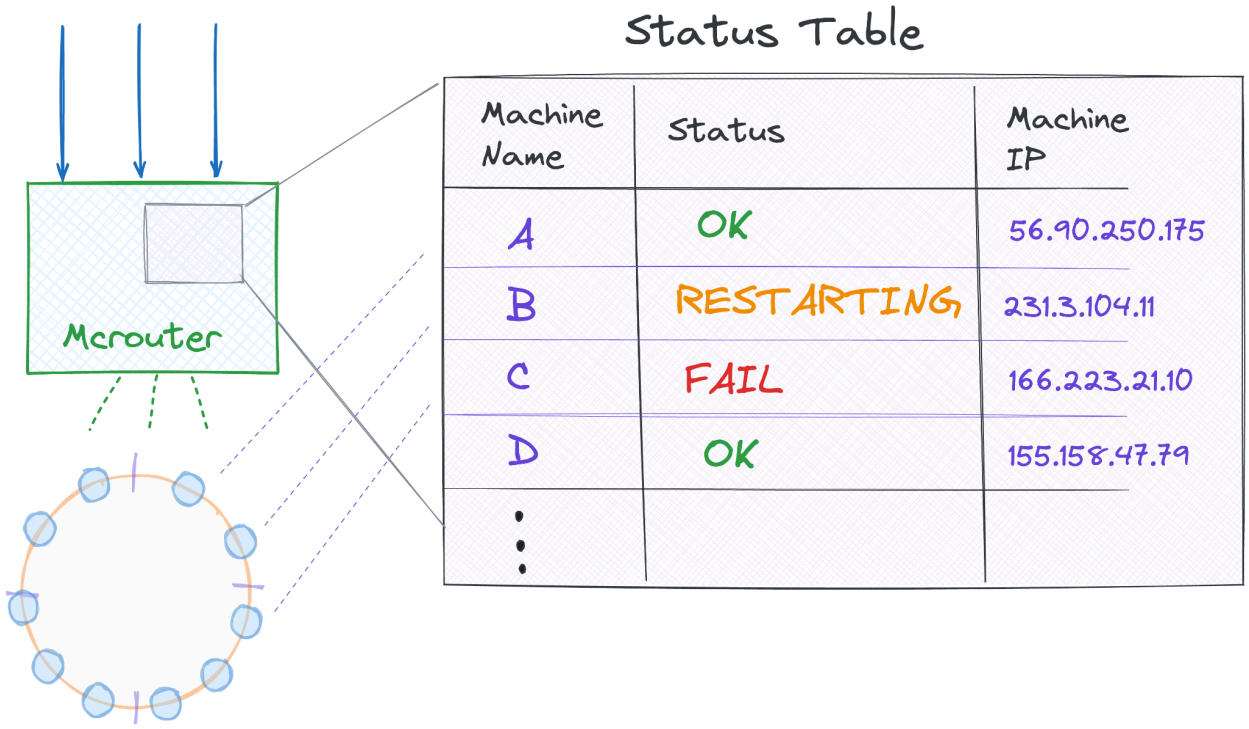
When we add a new machine to the ring, we update the configuration in Zookeeper, which then updates the rest of the Mcrouter servers. This ensures all Mcrouter instances have the latest configuration.



And that's it! We've built an extremely scalable routing system that can handle thousands of machines efficiently. This system ensures that all changes are propagated quickly and accurately, keeping everything in sync and running smoothly.

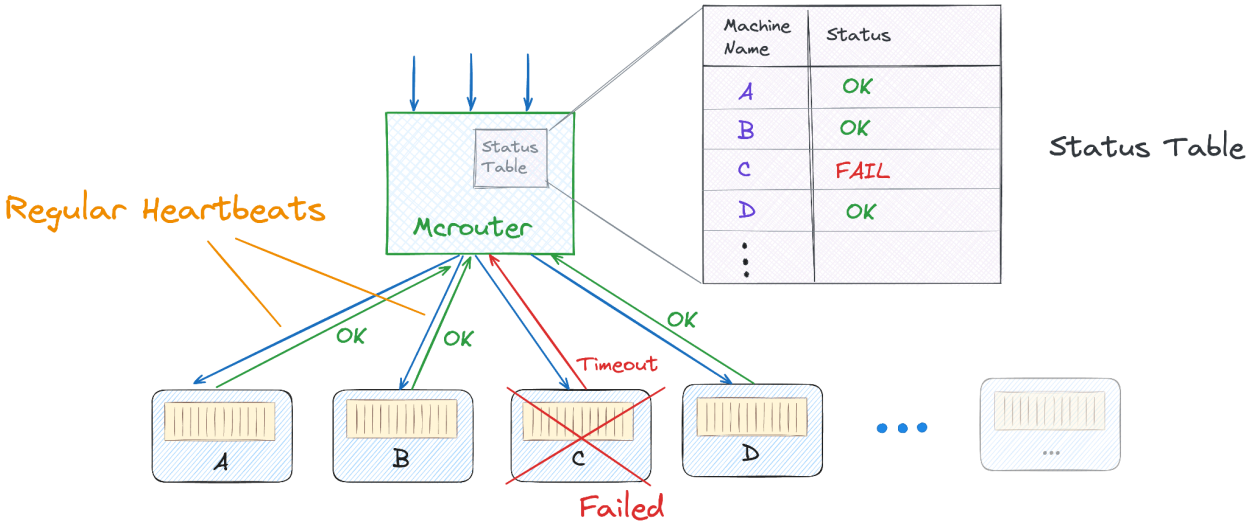
Heartbeats

Mcrouter keeps track of all the servers it manages. It maintains a table for server status.



When Mcrouter sees that a server has stopped responding, it should redirect requests to the server's replica, if one exists. We'll talk more about replicas later.

Heartbeats mean that the Mcrouter instance sends out regular pings every second or so. If it doesn't receive a good response, it can mark the server as failed and take action. Heartbeats also serve another purpose: they can transport stats back to Mcrouter for monitoring since regular requests are going through anyway.



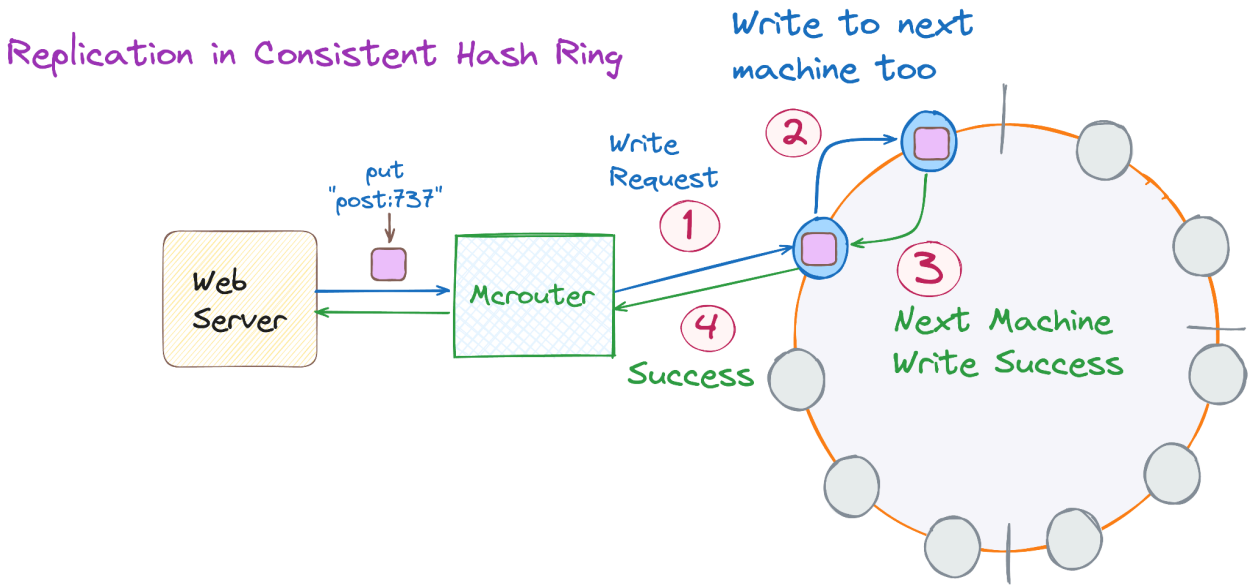
How to Handle Failures with McRouter?

Mcrouter can be designed to handle machine failures.

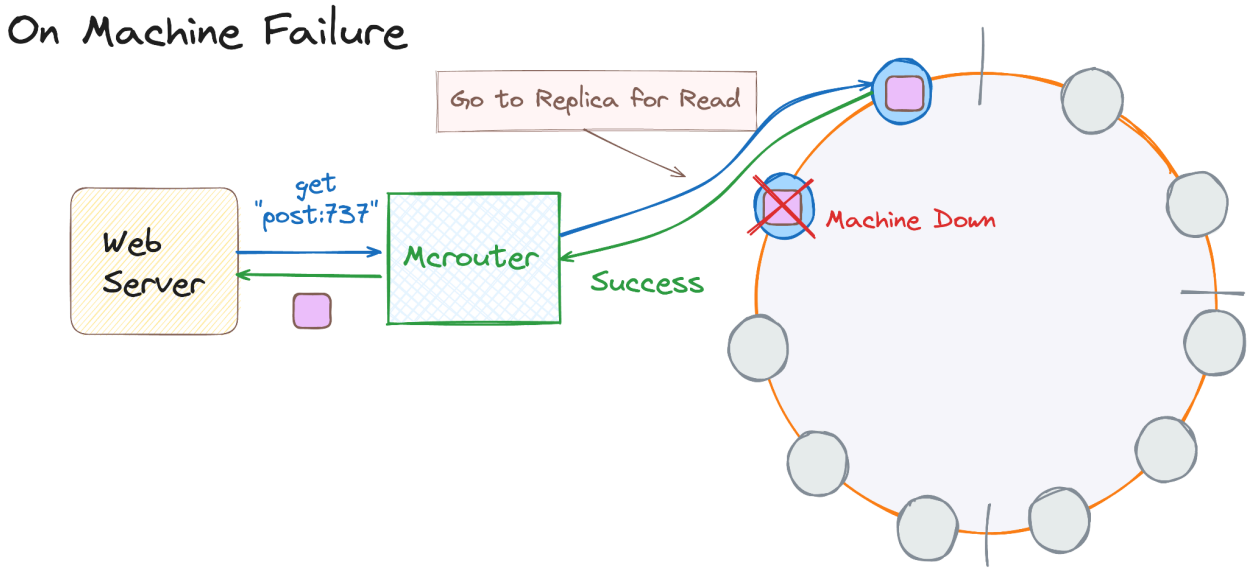
As soon as Mcrouter sees heartbeats missing, it knows that a server has failed. The steps it takes next depend on the system's configuration.

With Replication

If replication is active, multiple machines store the same keys. For example, "post:456" is written to 2 machines at once.

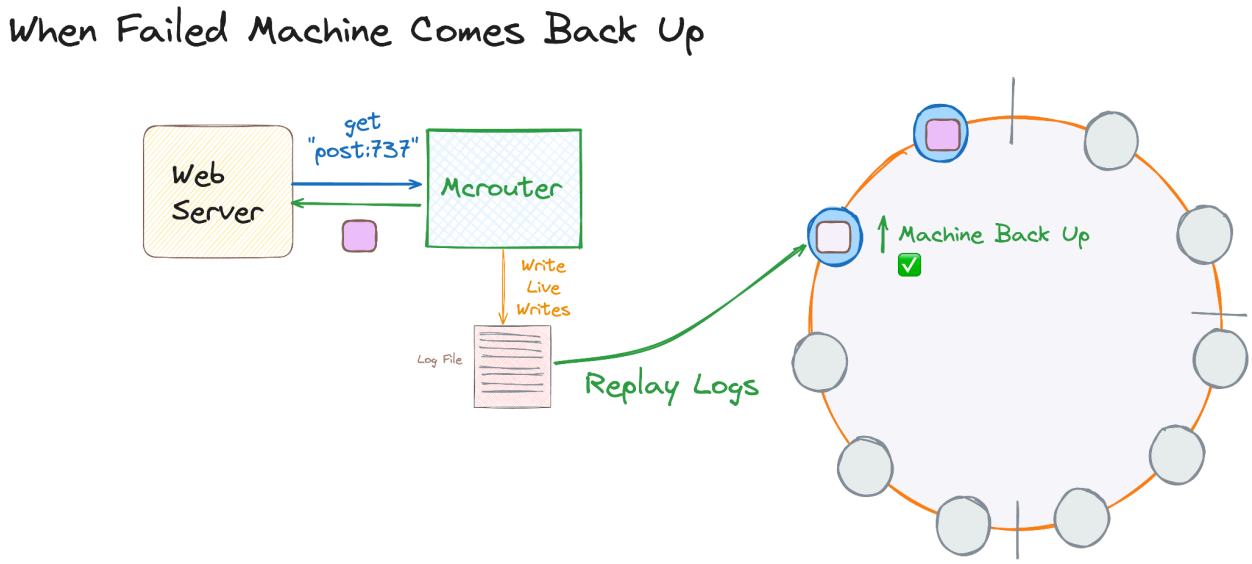


In this case, Mcrouter can forward read requests to the replica.



This part is simple. However, write and delete requests are more complex. These requests need to be replicated to the other machine when it comes back up. Mcrouter can log all the put and delete requests to a file.

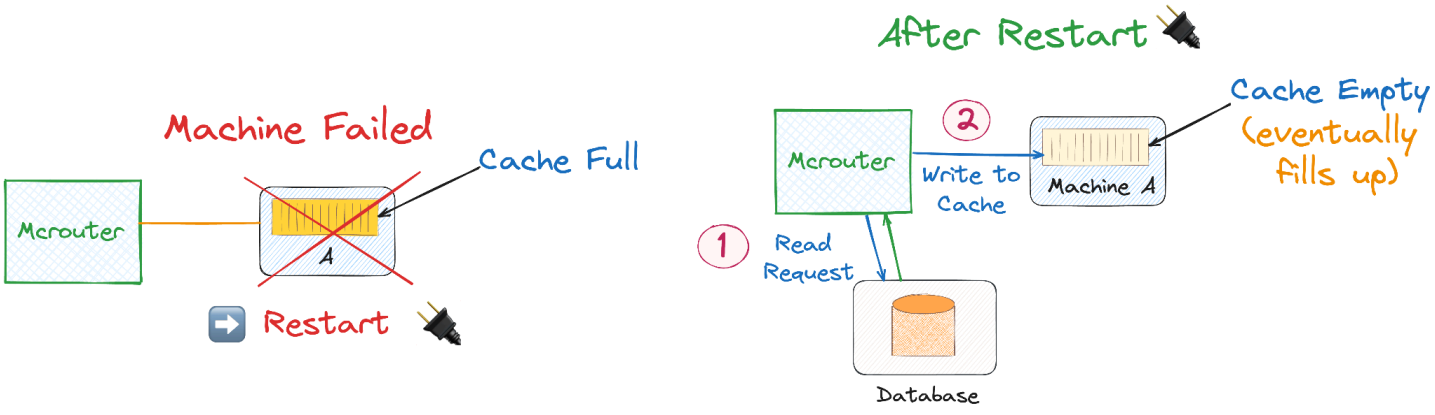
When the failed machine comes back up, it can replay those logs and restore itself to the latest state.



In this setup, we've minimized performance disruption. The only performance hit occurs if the replica cannot handle the extra traffic.

Without Replication

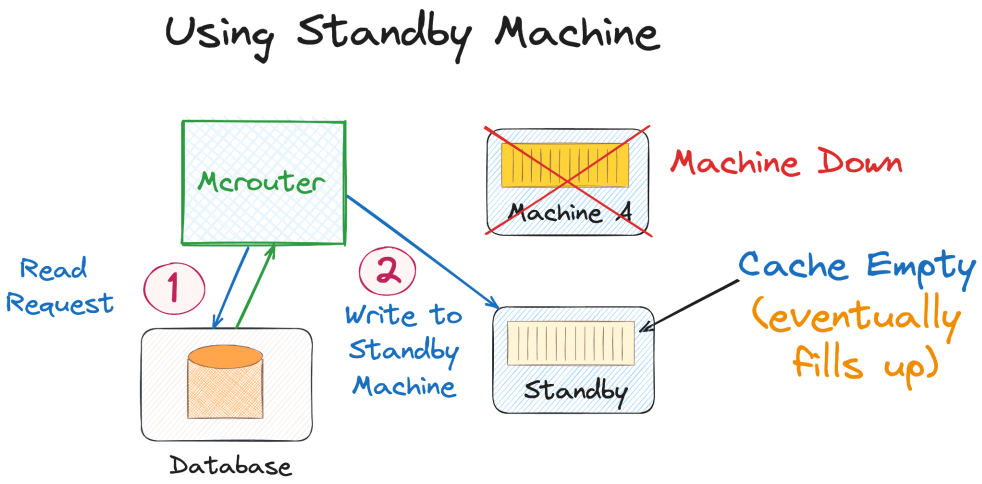
Without replication, our options change. The simpler option is to try restarting the server after waiting for it to recover. If the restart succeeds, all its data will be gone. However, we will continue writing to it when we see its keys missing in the cache. Soon, it will fill up like before.



If the server restarts within a short time, like 5 minutes, a few users might see slow queries for a while, but that's it.

We can handle it better. Can you think of a way?

It's simple. We have a standby machine ready for such situations.



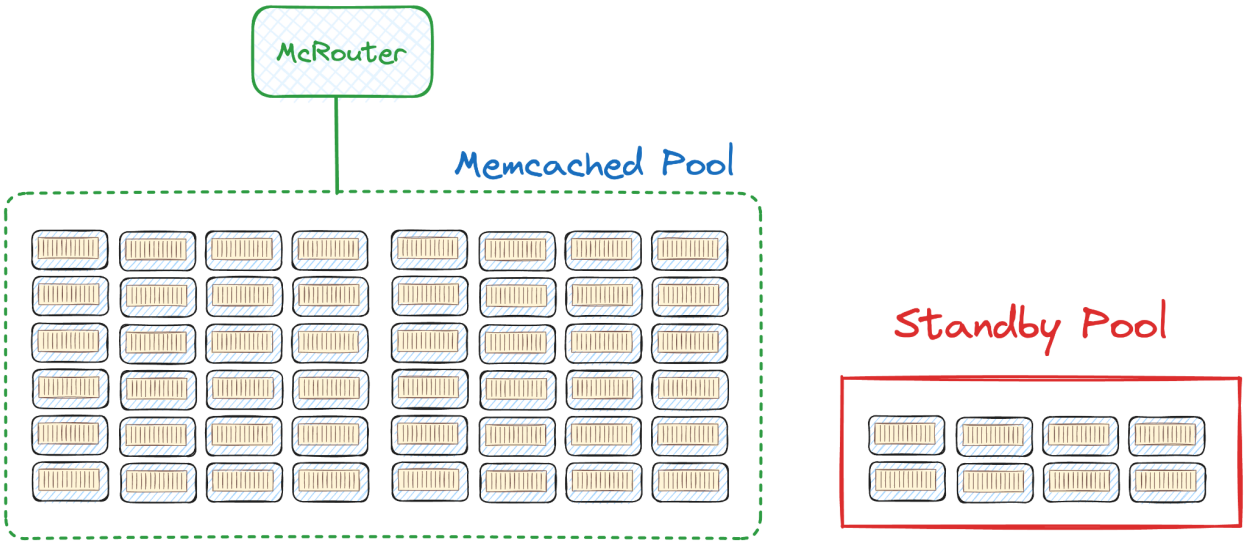
When we detect a machine failure, we start directing that machine's requests to the standby. There will be cache misses at first, but it will fill up quickly and continue as usual. Meanwhile, if the original machine comes back up, we have two options:

1. **Designate it as the new standby**: This requires no extra work.
2. **Copy data from the standby back to this machine**: This gets the original machine back into the active pool.

Option 1 is better because it requires no extra work.

Standby Pool

We can have a standby pool within our cluster. If we have 100 machines in our memcached cluster, we can dedicate 10 machines as standby. This way, even if 10 machines go down at once, we'll be fine!



How do you treat Soft Errors vs Hard Errors?

Soft Errors are temporary issues that might resolve on their own. For example, temporary network congestion can lead to a request timeout.

With soft errors, we can retry the request and give the server time to recover.

Examples of soft errors include:

- **Network Issues**: Temporary congestion, packet loss, or other network-related problems.
- **Server Issues**: Temporary overload, where the server is too busy but might soon be back to normal.

If the server or network doesn't recover within a certain period, we should treat it as a hard error and divert requests to a backup machine.

Hard Errors are issues that require immediate action. For instance, if we get a "connection refused" error, it means the server is completely down. This could be due to:

- **Hardware Failure**: The machine has crashed or there is a physical issue.
- **Software Crash**: The server software has failed and is not running.
- **Critical Errors**: Any other severe error that causes the server to stop responding.

When we encounter hard errors, we don't wait for recovery. We immediately redirect requests to a backup machine to maintain service availability.

In summary, for soft errors, we allow time for recovery and retry requests. For hard errors, we act quickly to switch to a backup, ensuring minimal disruption to the service.

Recap

Let's review what we have so far. We've set up a cluster of 100 machines, managed by Mcrouter. Here's what we've accomplished:

1. **Request Routing**: We've learned how to route requests using consistent hashing. This ensures that each request goes to the correct machine based on the key.

2. **Scalability**: We can add more machines to the ring when needed. This allows our system to grow and handle more traffic without major disruptions.

3. **Heartbeats**: We keep heartbeats to each machine in our cluster. These regular pings let us know if a machine is still alive. If a machine stops responding, we can detect the failure quickly.

4. **Failure Recovery**: We've figured out how to recover from failures. If a machine goes down, we can redirect requests to a backup machine or use a standby machine to take over the load.

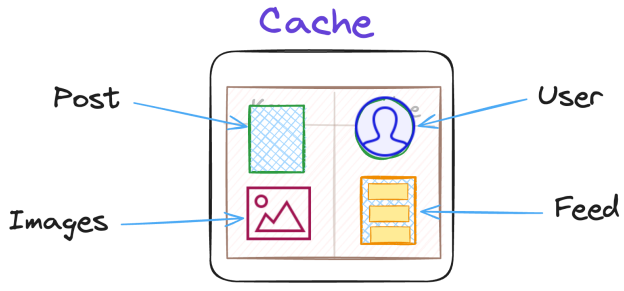
We're doing well! We now understand how to run an entire cluster effectively.

Now, let's add some advanced features to make our system even more robust and efficient.

Ch 7. Optimizing with Memcached Pools

How to solve issues by using Special Pools?

So far, we've built a large cluster with hundreds of machines. However, remember when we talked about how memcached doesn't have separate "tables" like databases do? Everything is pushed into a common table, whether it's a "user", "image", or "post".



Each machine in our cluster is like a mixed salad – it has everything mixed together.

What problems can you predict with this setup? Think about it and jot down a few ideas.

The main problem is this: certain types of data may interfere with other types of data. For example, data with lots of keys (like user session data) might push out data with less keys (like popular posts) causing cache misses for important data.

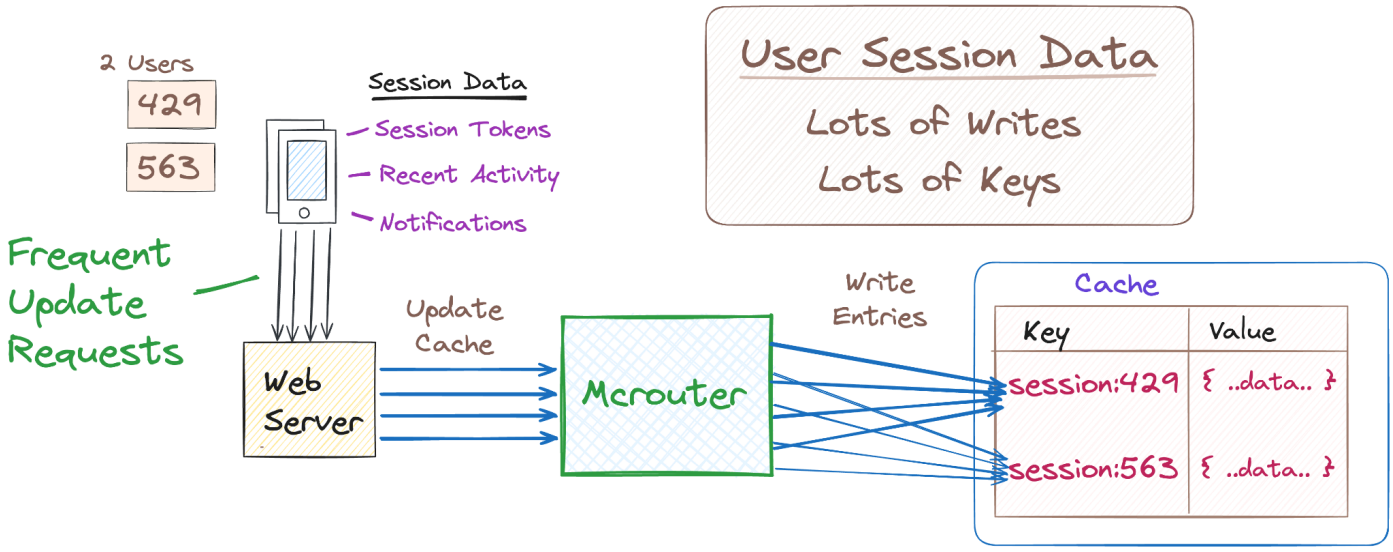
Another problem is that certain data types might have specific requirements. For instance, some data might need enhanced security, while other data might need replication to ensure high availability. Mixing everything in a single table can make it difficult to meet these specific needs.

Data That Interferes with Other Data

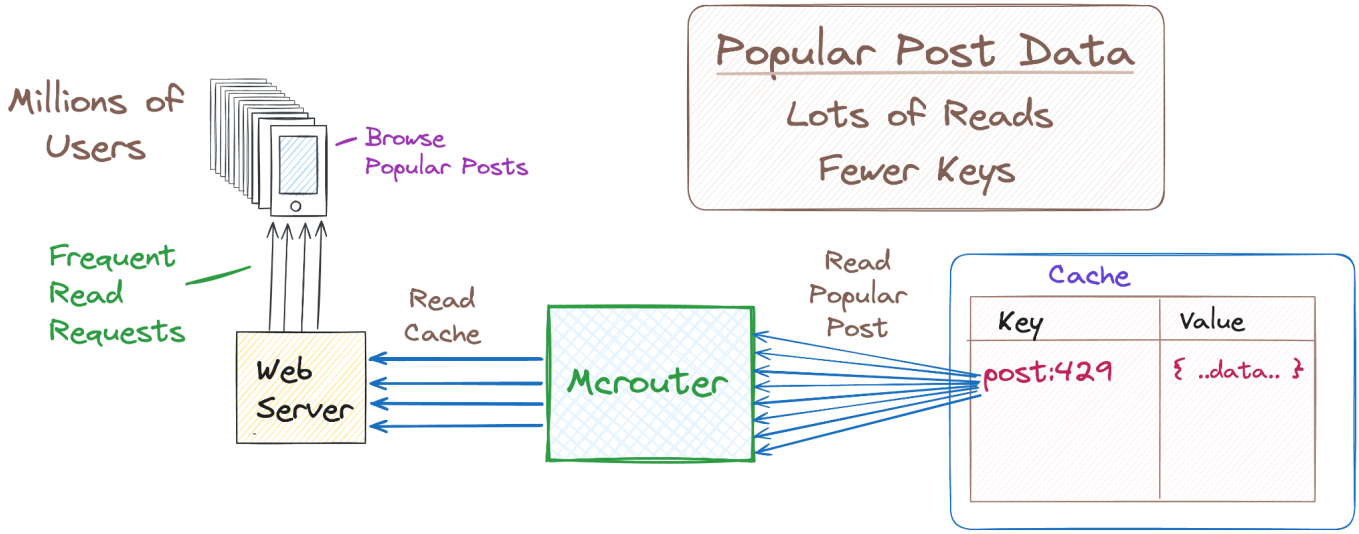
Let's look at the first issue: data that interferes with other data. Let's take two types of data:

1. **User Session Data**: This is your temporary, user-specific info—login status, what posts they liked, what comments they made. It's like your app's memory for each user, always changing.
2. **Popular Posts**: These are the trending topics, viral videos, and top discussions everyone's talking about. They're accessed by everyone, but they don't change as often.

User session data is a whirlwind, constantly updating as users interact with the app. Every like, every comment, it's all written to the cache. While using the app, this data is constantly read by the user. And it's a lot of data - one key per user.

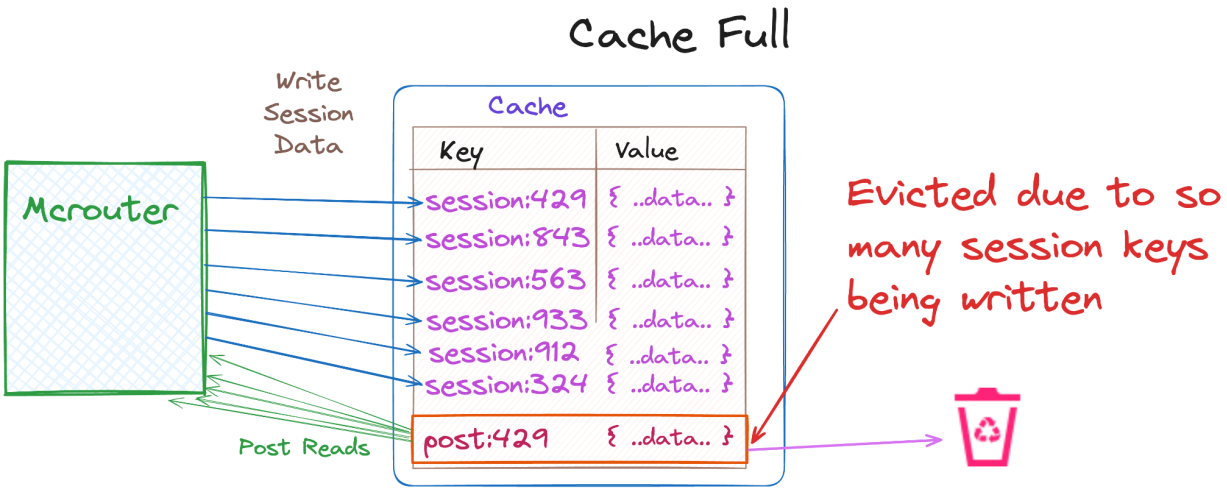


Popular Posts, on the other hand, are written once, and read by a huge number of people.

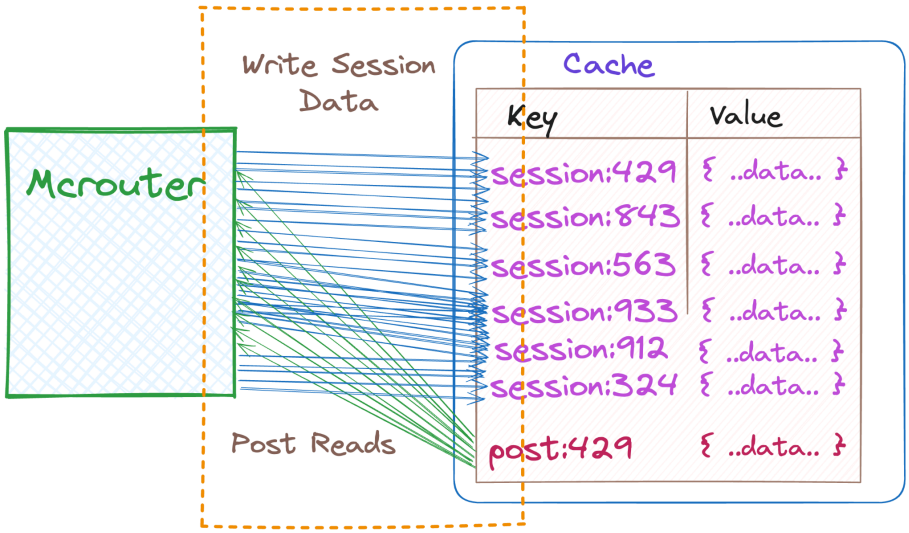


When these two types of data share the same Memcached pool, you run into issues:

1. Frequent Evictions: All that volume of constant session data can evict your popular posts. The next time someone wants to see that viral video, it has to be fetched from the database again, slowing things down.



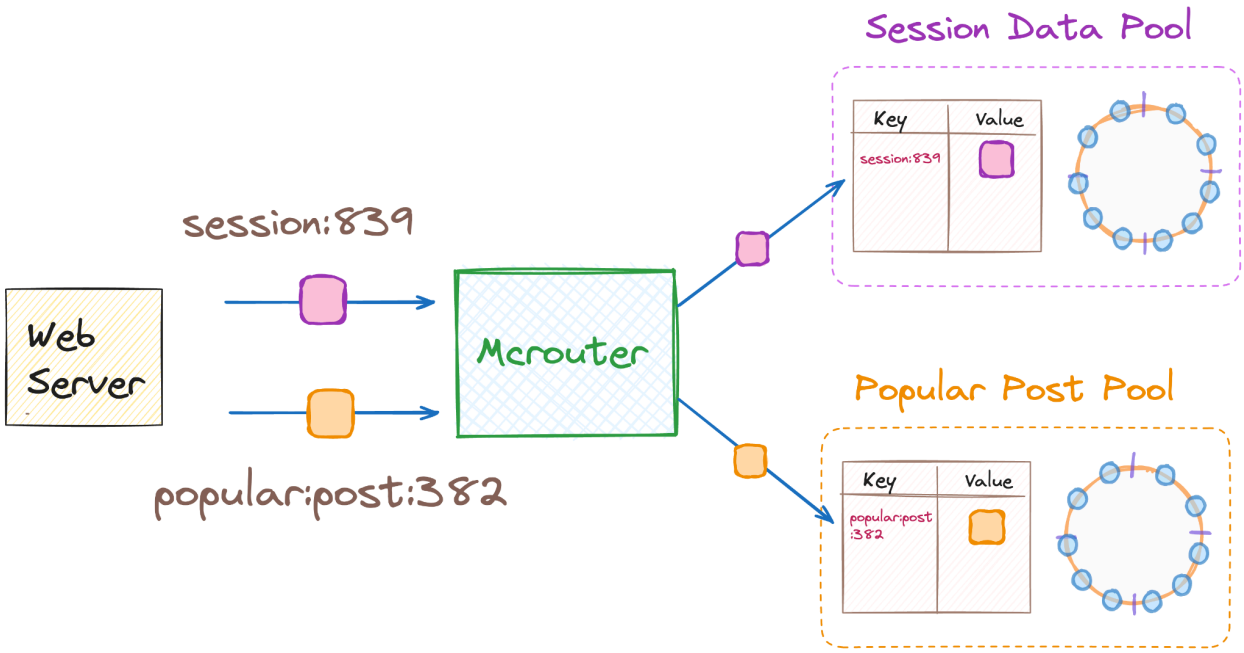
2. Resource Contention: Different needs (frequent writes for sessions vs. frequent reads for posts) mean both types of data are competing for cache resources, causing performance issues.



Solution: Using Separate Pools

Here's the solution: Create separate Memcached pools for different data types.

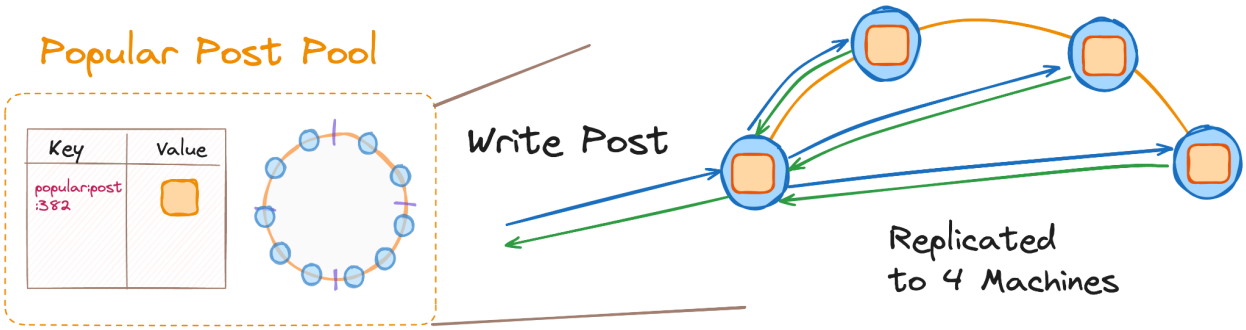
1. Session Data Pool: This pool is dedicated to user sessions. It's optimized for high write throughput and has eviction policies suited for session data.
2. Popular Posts Pool: This one's for your popular posts. It's optimized for read-heavy workloads (it has read replicas) and has longer TTL settings to keep the popular posts in the cache longer.



Special Replicated Pool

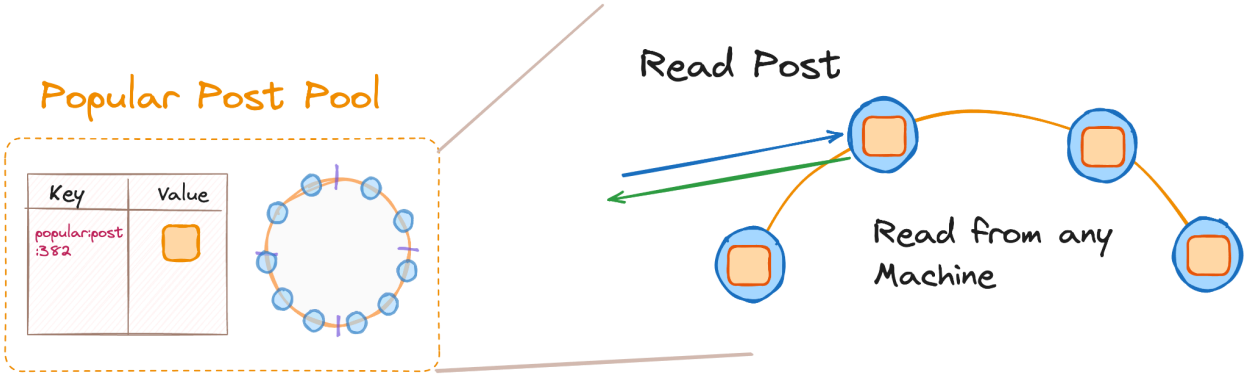
To handle heavy reads, we can create a special pool with read replicas for every key. Here's how it works:

- Replication: Each write goes to, let's say, 4 machines in this special pool.



- High Throughput: This replicated pool can handle much higher read throughput.

- Load Distribution: It shelters the common pool from the heavy load of popular posts.



It's like having a cache for our cache! Our main cache protects our database from load, and our special pool protects our main cache from being overwhelmed by high-traffic data.

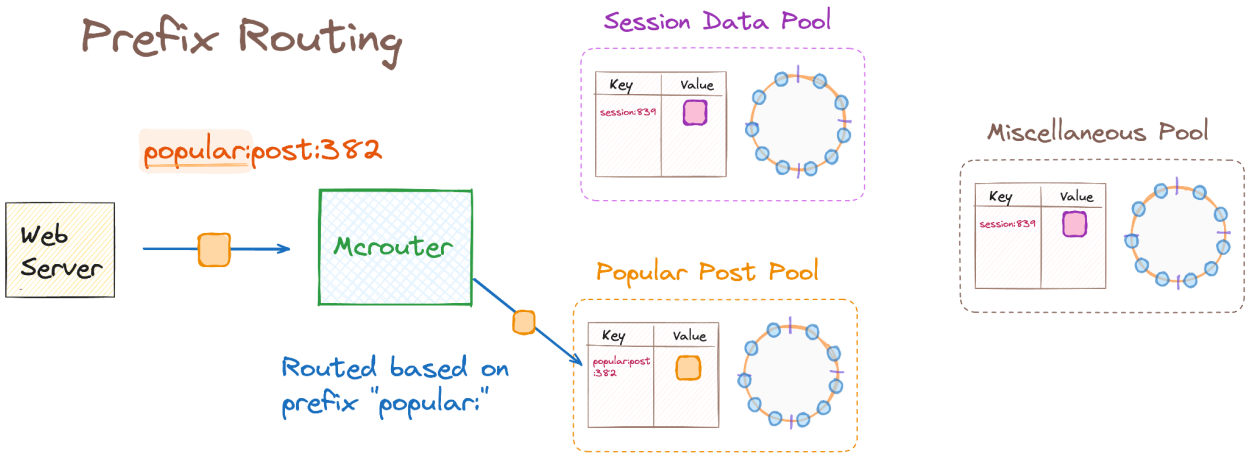
By creating different pools, we can make that each type of data gets the treatment it needs. This makes our system better suited to handle diverse workloads.

How can you Tag and Route Popular Posts?

How does Mcrouter know to put a post in the special pool?

Our server will need to tag a popular post with a prefix. For example, let's say post ID 763 is popular. Our backend has a system for determining this. As part of the post JSON properties, we have a flag: `cache-popular: true`.

When our web server sees this flag, it adds a prefix "popular:" to our normal "post:763" key. So now the key becomes "popular:post:763". We configure Mcrouter to redirect that prefix to the "popular" pool.



Now, when the post is fetched, it is fetched from the popular pool, where we can handle a lot of throughput. The rest of the cache remains untouched.

We've covered three crucial concepts here:
1. **Memcached Pools**: Different groups for different types of data to optimize performance.
2. **Replicated Pools**: Special pools with replication to handle high read loads.
3. **Prefix Routing**: Using prefixes to route specific keys to designated pools.

By implementing these strategies, we can manage and scale our caching system to handle high traffic and different data types.

With these approaches, our memcached setup should scale like a charm.

But wait, there's one more level of scaling we haven't covered yet: interregional memcached. This is where things get really interesting—managing caches across different regions and data centers.

We'll dive into that later. Stay tuned!

..More Chapters Coming Soon

There's many more things I'd like to cover. How to scale globally across continents. Strategies for eviction. How to optimize the cache hit ratio. I'll keep adding more chapters. Thanks for your patience and for getting this far 🙏.

To get new sections and updates, please visit harsh.fyi/subscribe. I plan to keep sending updates on where I am and new sections I am working on.

References

https://www.linuxjournal.com/article/7451
https://research.facebook.com/publications/scaling-memcache-at-facebook/
https://medium.com/pinterest-engineering/scaling-cache-infrastructure-at-pinterest-422d6d294ece
https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/WhatIs.html
https://docs.aws.amazon.com/whitepapers/latest/database-caching-strategies-using-redis/caching-patterns.html
https://swetava.wordpress.com/2018/10/21/latency-numbers-every-programmer-should-know/
https://www.mikeperham.com/2009/06/22/slabs-pages-chunks-and-memcached/
https://docs.oracle.com/javase/tutorial/networking/sockets/clientServer.html
https://www.digitalocean.com/community/tutorials/memcached-telnet-commands-example
https://stackoverflow.com/questions/6553067/hashtable-vs-hashmap-performance-in-single-threaded-app
https://stackoverflow.com/questions/3554888/what-is-the-purpose-of-colons-within-redis-keys
https://stackoverflow.com/questions/1378310/performance-concurrenthashmap-vs-hashmap
https://www.dragonflydb.io/faq/what-happens-when-memcached-is-full
https://engineering.fb.com/2014/09/15/web/introducing-mcrouter-a-memcached-protocol-router-for-scaling-memcached-deployments/
https://stackoverflow.com/questions/10935231/multiple-caches-in-memcached
https://linux.die.net/man/1/memcached

[1] Here is the famous "Numbers Everyone Should Know" slide by Jeff Dean.