**Lecture 10: Hashing**

We will discuss some simple hashing and some techniques such as

Consistent Hashing

Rendezvous Hashing

Hashing is an action that we can perform to transform arbitrary piece of data into a fixed size value, typically an integer value.

In context of system design interview, that arbitrary piece of data can be an IP address, username, HTTP request or anything that can be hashed into an integer value.

Suppose we have 4 clients c1, c2, c3, c4 which will be speaking with 4 servers A, B, C and D. And there is a load balancer in the middle.

When a load balancer has to re-direct a request from a client to a server, it has to have some sort of server selection strategy, that is to pick the server to send the request to.

While a lot of the server selection strategies are fine at there face values, they can introduce some problems depending on our system.

Suppose the requests being made by the clients are computationally expensive. When the servers get these requests they have to perform some very complex operations that take a long time to complete and these requests are just expensive.

One way we can use to deal with these types of requests in a system is to use caching.

We can make the servers store an in-memory cache where every request that goes through a server, the server first checks if that request’s response has been cached.

If that hasn’t been cached then it actually performs the operations and gets the response but if it has been cached then it skips the operations that it would otherwise have to do and immediately returns the cached response.

That would be a reasonable way to design our system if we knew that the requests would be computationally expensive.

If the load balancer is selecting servers to re-route requests to following a round-robin or random strategy, this is where we are going to have problems.

Suppose if client 1 sends a request and that request has to be re-directed to server A, the load balancer follows a round robin approach and this time around it re-directs the request to server A.

Server A checks its cache, lets say that the cache is empty, so the server A performs the long operation, then it stores the response in the cache.

The response of the request from client 1 is now in the cache and then the response is returned back to the client.

Now the client 1 issues the same request once again. Now it happens that with the Round Robin approach, the load balancer re-directs the request to server D.

Server D has never got the request from client 1 before. The server D does not have the response to this request in its in-memory cache.

The server A has the response to this request but not server D.

This is where we get a big problem.

If we have designed our system in such a way that it relies on in-memory caches in its servers but our load balancer is redirecting requests from clients following a server selection strategy that does not redirects requests from a single client or the same requests will be re-routed to the same server every time, then our

in-memory caching system is gonna fall apart and it is not going to be nearly as useful as it could be otherwise.

We will miss a couple of cache-hits when our requests will be routed to a server that does not have responses cached.

This is where hashing comes into play.

With hashing, we can hash the requests that come into the load balancer and based on the hash we can bucket the requests according to the position of the servers.

Once we have hashed them, we can bucket the requests in the servers by performing a tiny bit of business logic.

For simplicity, let’s just hash the names of our clients.

Our goal is to basically get every client to have all of its requests re-routed to the same server.

Lets assume that we have been given a hashing function and when we pass c1, c2, c3 and c4 through that hashing function, we get the following results, 11 for c1, 12, 13, 14 respectively.

We can take the modulus of the hash values with the number of servers we have.

After taking modulus, we get numbers corresponding to the 4 servers that the 4 clients should be associated with.

When we are dealing with a hashing function, it is important that the hashing function have uniformity.

Our clients got evenly distributed amongst the four servers. That did not necessarily happen by chance. Our clients got evenly distributed among our 4 servers.

That did not necessarily happen by chance. We chose the integer values that made them to distribute evenly amongst the servers.

A really good hashing function would also be able to do this as well. It will give us this sort of uniformity.

We could have the case of 2 clients mapping to the same server.

The point is that a good hashing function would evenly distribute our data values. That is something to keep in mind when we are dealing with hashing.

In practice, we typically never write our own hashing function. We use a pre-made, industry grade hashing function in our hashing algorithm.

NV5 hashing algorithms

SHA-1 hashing algorithm, bcrypt hashing function

We can trust that these hashing functions have uniformity about them.

Now, when client c1 issues requests, they are always going to go server D.

If server D has an in-memory cache, we will always get cache hits.

We will maximise the chances of cache hits for client c1’s requests.

We fixed the problem that we described before where we were gonna miss our cache hits because of the server selection strategy of our load balancer which would make our client requests potentially hit different servers everytime.

But, we have another problem now.

We are dealing with a large scale distributed system. And when we have such a system, a lot of things can happen.

For instance, our servers can die. We could very easily have server A fail on us and completely die.

Similarly our system could be experiencing a ton of traffic and we might need to add servers to our system.

Maybe our server A does not die, maybe we need to add a new server to our system, server E to handle some new incoming traffic.

**What do we do if need to add a 5th server ?**

This is not an unreasonable thing to expect if we are dealing with a large scale system.

What do we do if we need to add server E ?

We cannot continue to mod all of our hashes by 4 because we no longer have 4 servers. We have 5 servers.

If we keep modding the hashes of our clients by 4, then all the requests will always going to be redirected to servers A, B, C and D.

They are never going to go to E.

Before server A had died and we had not added E, then we would not have been able to continue to mod by 4 because requests would have continued to get routed to A and A would have been dead and our users would not be happy. (Server A is dead and E is not there).

If a server dies or if we add a new server, we have to change some logic in the hashing.

Mainly we have to mod our hashes by the new number of servers.

We are once again going to face problems of missing some cache hits. The in-memory values that we had in our caches will be lost.

What can we do when we have to add a server or when we have to remove a server ?

This is where Consistent hashing and rendezvous hashing come into play.

**Consistent Hashing**:

Instead of imagining our servers in a vertical line, we are actually going to be put them in a circle.

It will help us to visualize consistent hashing better.

Suppose we have placed the servers on the circle in an evenly distributed way.

Suppose the circle represents numbers back to back with 0 at the top and increment as we move along the circle in a clockwise way all the way until we reach the top again.

The circle can be represented as a set of back to back numbers.

Conceptually, it looks like a circle that loops around.