Consistent Hashing

Consistent hashing is a strategy for dividing up keys/data between multiple machines.

It works particularly well when the number of machines storing data may change. This makes it a useful trick for system design questions involving large, distributed databases, which have many machines and must account for machine failure.

A Hash Map With *Machines* Instead of *Buckets*

Conceptually, consistent hashing has a lot in common with hash maps.

The main difference is that instead of mapping a key to a value, we'll be mapping a key to the machine responsible for storing the value.

Just like a hash map, we'll start out by taking each key we want to store and hashing it (/concept/hashing) to get a **hash value**—usually, a number.

What does this **hash function** look like?

There are a lots of implementations: MD5, SHA1, and SHA256 are some of the common ones. You'll want to use one of those, instead of coming up with your own.

Each hash function produces a *range* of values. As an example, hashing with MD5 gets you a value between 0 and $2^{128} - 1$. (That's 340,282,366,920,938,463,463,374,607,431,768,211,455, in case you're wondering.)

We can visualize this as a number line:

The next step is to take this big number and translate it into a machine. Visually, we're dividing up the number line into chunks, and assigning each chunk to a machine.

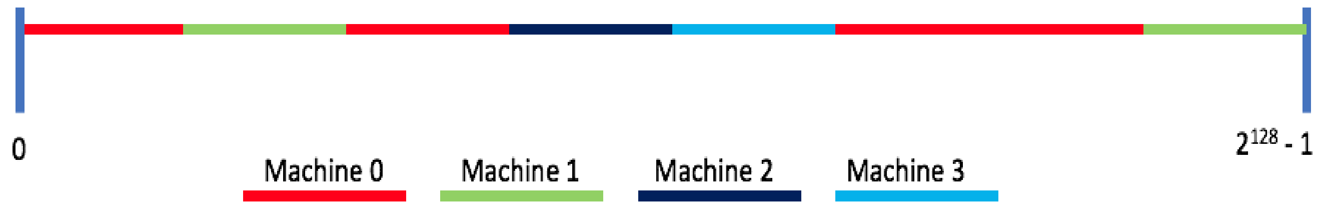


IMAGE EDIT: Machine A, Machine B, Machine C, Machine D.

The secret sauce in consistent hashing is *how* we divide up the number line.

Dividing hash values between machines

Ideally, we want a division strategy that lets us add or remove machines without reassigning lots of keys.

> Remember, the big benefit of consistent hashing is that it can handle machines being added or removed. That may seem like an edge case, but in large systems with thousands of machines, failure is normal.

Let's compare a few different options:

Option 1: Mod the hash value by the number of machines.

With this strategy, if we have N machines, we compute $hash$ to figure out the machine responsible.

So, with 3 machines, a hash value of 0 would map to machine A, 1 maps to machine B, and 2 maps to machine C. If we keep counting up, we start to repeat machines. So, 3 maps to machine A, 4 maps to machine B, and 5 maps to machine C.

Visually, we've divided up our number line like this:

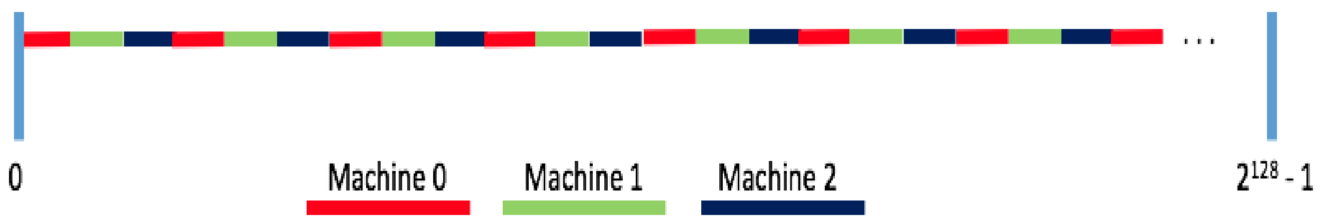


IMAGE EDIT: Machine A, Machine B, Machine C

Now, say we add another machine. Now, we're computing $hash$ instead of $hash$.

Here's how our assignments along the number line change:

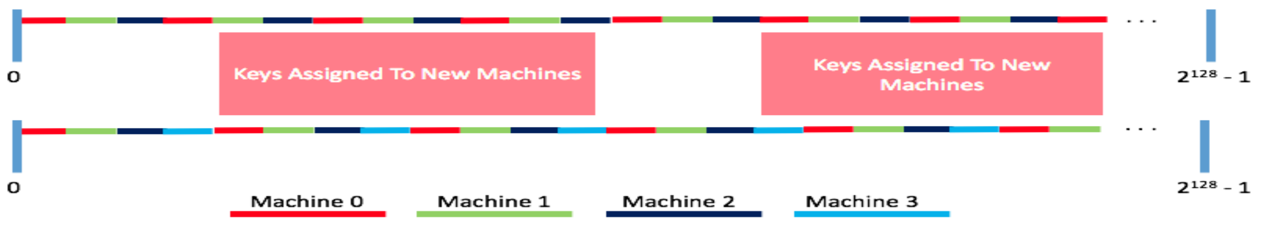


IMAGE EDIT: Machine A, Machine B, Machine C, Machine D.

Look at how our number line changes! Lots of keys are now assigned to different machines. To accommodate this, we need to send data from its old machine to its new one—that's lots of network traffic and could take a long time with large amounts of data.

Option 2: Divide the number line into N even chunks.

If we have three machines, our number line would look like this:

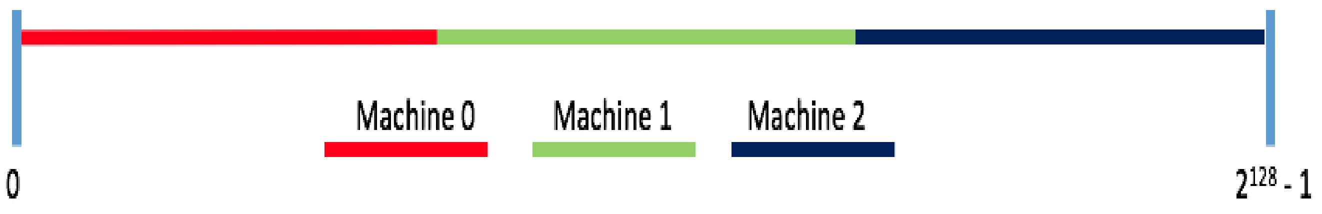


IMAGE EDIT: Machine A, Machine B, Machine C

What happens when we add another machine?

We'll have to re-divide our number line, taking a bit from each of the three old machines to give to the new one. Here's what that looks like.

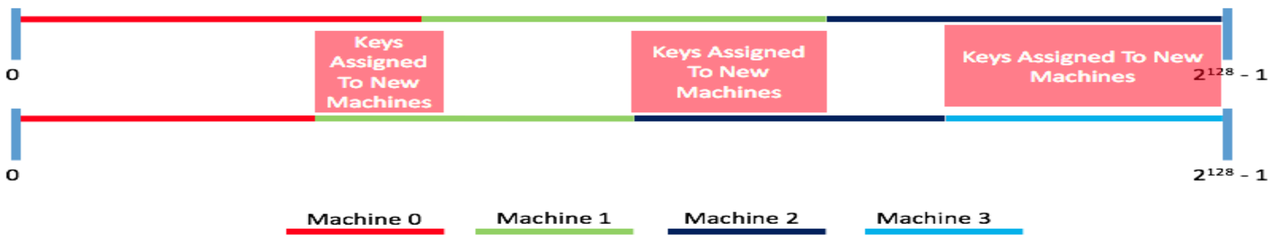

0
Keys Assigned To New Machines
Keys Assigned To New Machines
Keys Assigned To New Machines
$2^{128} - 1$
0
$2^{128} - 1$
Machine 0
Machine 1
Machine 2
Machine 3

IMAGE EDIT: Machine A, Machine B, Machine C, Machine D.

This seems like we're on the right track. We still have a lot of the number line that's been reassigned, but not as much as with the mod operator.

Option 3: Hash the *machines*, and assign keys to the *next* machine on the number line.

Here's what that looks like with 3 machines:

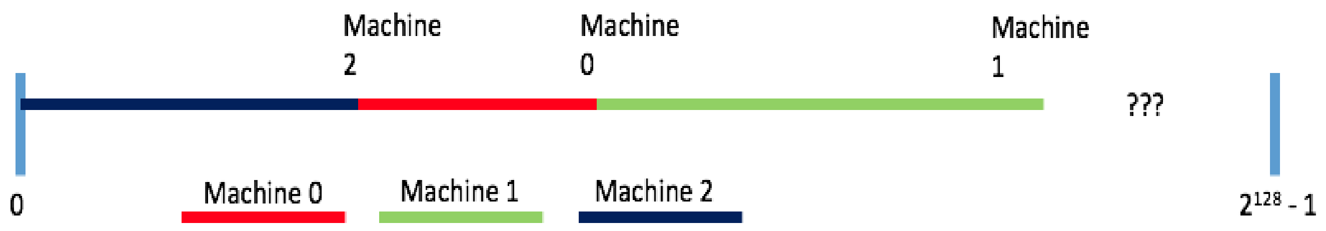


IMAGE EDIT: Machine A, Machine B, Machine C

There's a corner case though. What about numbers that appear above the highest-hashed machine?

We'll say that if you walk off one end of the number line ...

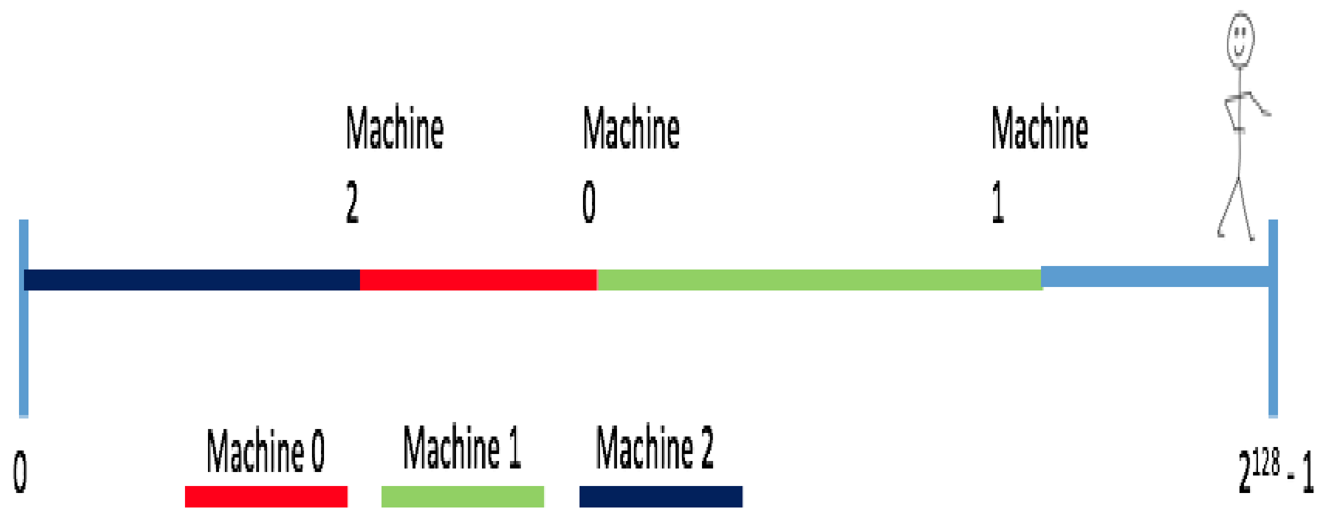


IMAGE EDIT: Machine A, Machine B, Machine C

... that you end up at the other end.

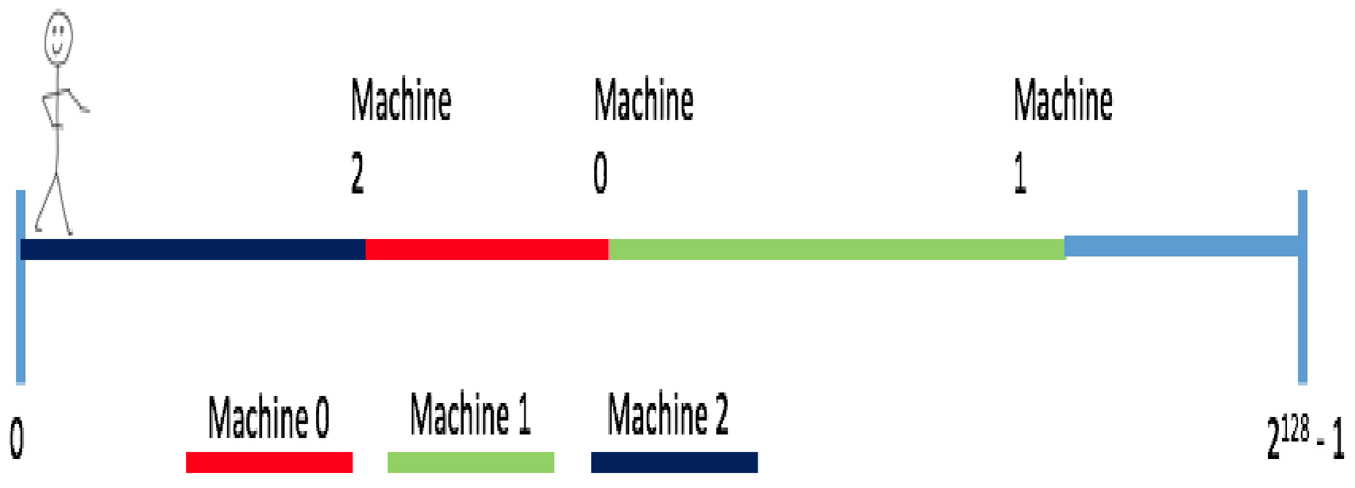


IMAGE EDIT: Machine A, Machine B, Machine C

Putting things together, we end up with a *ring*. Here's what it looks like with three machines.

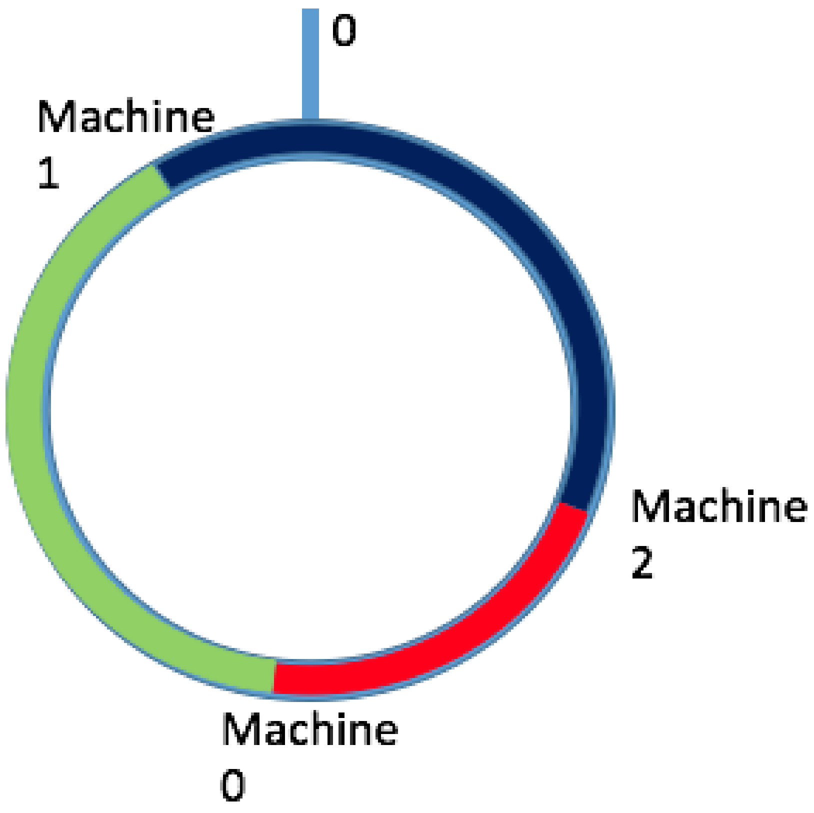


IMAGE EDIT: Machine A, Machine B, Machine C

Okay. What happens when we add a fourth machine?

It gets hashed, placed along the number ring, and takes some of the keys from the machine that appears after it.

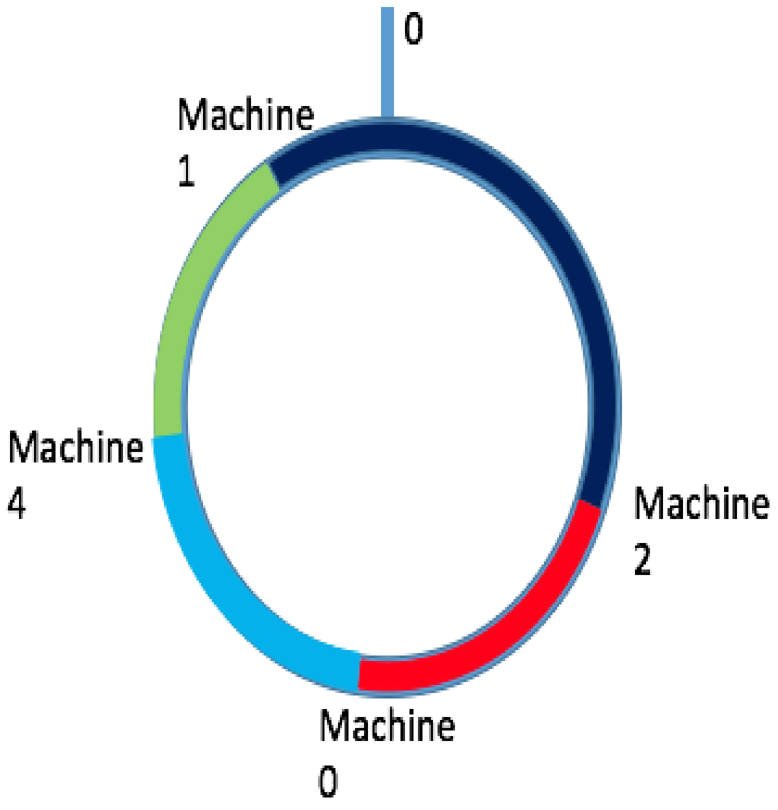


IMAGE EDIT: Machine A, Machine B, Machine C, Machine D.

Nice. We're not reassigning lots of data. In fact, only one machine needs to send data to the new machine.

This is the strategy used by consistent hashing.

Like we said above: **the secret sauce of consistent hashing is how we assign keys to machines**. Our assignment strategy works well for large systems where the number of machines can change, because it doesn't reassign too many keys every time we add or remove a machine.

What happens if the machines aren't evenly spaced around the ring?

Here's what this looks like:

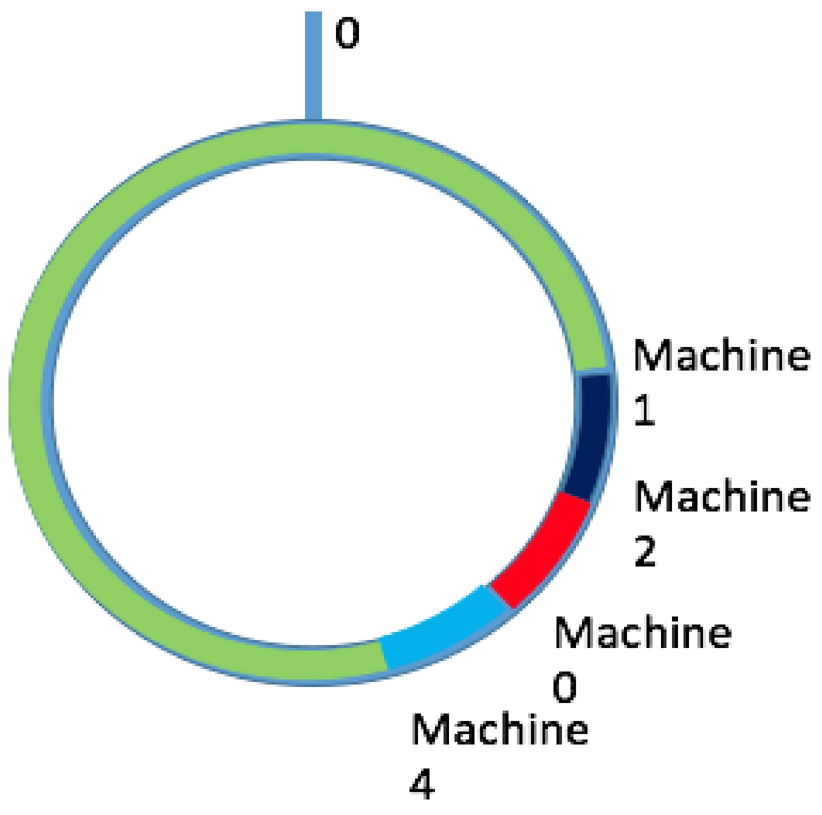
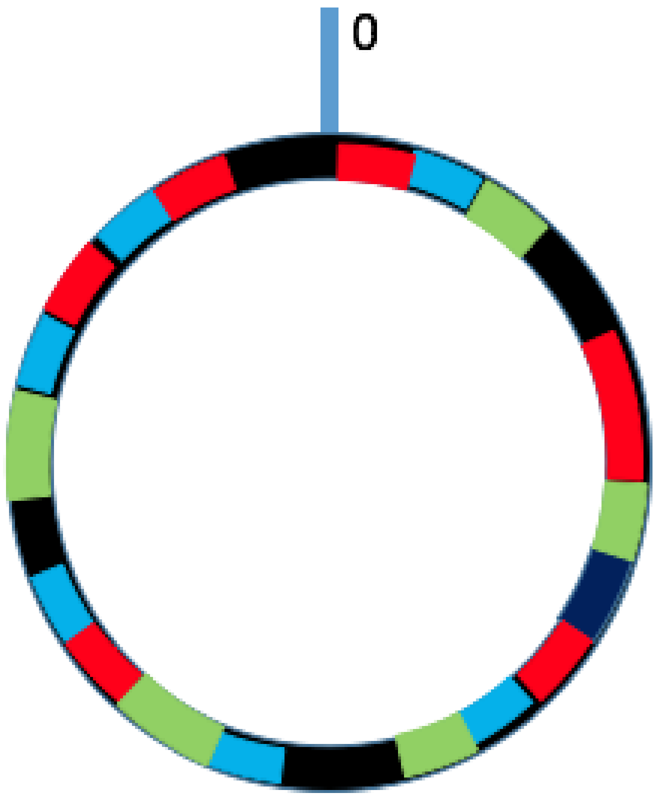


IMAGE EDIT: Machine A, Machine B, Machine C, Machine D.

That's not ideal. One machine has *way more* keys than the others!

To avoid this, it's common to place each machine along the ring multiple times. Usually, that means giving machines multiple identifiers and hashing all of them.

Here's what that might look like:



Usually, we'll leave these extra placements out of drawings showing consistent hashing, since it can make the picture kind of messy. This is the sort of tidbit you should mention to your interviewer but probably don't need to draw out on the whiteboard. :)

> This looks a lot like the first option. Is it any better?
>
> It is! With the first option, when we added a new machine, a majority of the keys were reassigned to new machines. With this division strategy, the total number of keys reassigned is much lower (only $\approx 1/n$ of the keys).

What happens if a machine leaves?

With consistent hashing we're assuming that machines can leave and join over time. If a machine leaves, won't we lose data?

To avoid this, we'll usually have machines act as backups for each other. One strategy is to have each machine replicate the data stored on the machine behind of it, giving us a backup copy.

Again, this is the sort of implementation detail you'll want to mention to your interviewer, even if you won't draw out the entire implementation.

What's next?

If you're ready to start applying these concepts to some problems, check out our mock coding interview questions (/next).

They mimic a real interview by offering hints when you're stuck or you're missing an optimization.

Try some questions now ➜