**Consistent Hashing**

Some of the things that we'll cover in this module are:

1. What is Hashing
2. What is Consistent hashing
3. Why is it important to know about Consistent hashing

What is hashing

**Hash is basically a function that takes a value and converts it into another value of a specific format.**

For example, if you have a list of users with certain ages and you want to sort them into buckets based on their ages, you can use a hash such that:

hash(age) = (age / 10) + 1

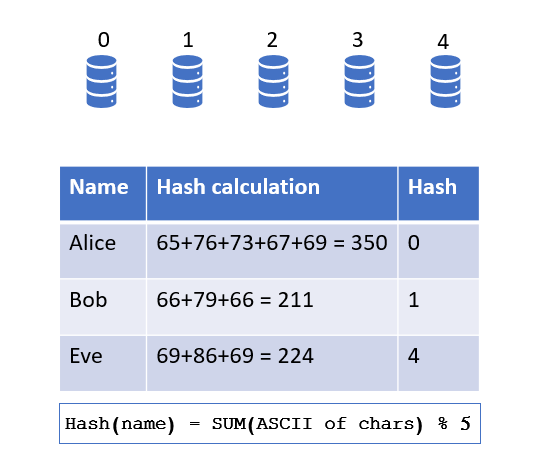
This is a very simple hash function. Some more realistic hash functions that are more commonly used are MD5, SHA1, SHA256, etc.

Why hashing?

Suppose you build a distributed cache, where the data is distributed over various nodes, sometimes spanning multiple data centers. When we want to store the data for a user, we need to decide which node will cache this data. And when we want to retrieve this cached value, we must query the same node. So let us use a simple hash function for this again.

hash(user) = Sum(ASCII value of all characters in name) % 5

Where 5 is the number of nodes. This way, the hashes generated for users will be as follows:



Since hash(Alice) is 0, Alice’s data will be stored in node 0. Similarly, Bob’s data will be stored in node 1 and Eve’s in 4. Now when you need to look up the information for Alice, you will use the same hash function to determine which node to query. Since hash(Alice) equals 0, you will query node 0 and fetch the information.

But there is a problem with this solution. This model is not scalable. If the traffic increases and you want to add a new node, the formula to calculate the hash will get updated to

hash(user) = Sum(ASCII value of all characters in name) % 6

And similarly hash(Alice) will get updated to 2. So now you will search for Alice’s information in node 2, but the information is actually stored in node 0 so you won’t find it.

To fix this, you will need to rehash all the data every time a node is added or removed, and once rehashed, you need to move the data to their respective new nodes, which could be across different data centers. This is not a very good solution as it uses up a lot of CPU resources and bandwidth.

This is where Consistent Hashing comes in. Consistent Hashing tries to optimize the system in such a way that:

* You don’t need to move around all the data while adding or removing nodes.
* There will be minimal movement of data as if a node is removed, the data from that node will be supported by another node. Similarly, when a node is added, some data will be mapped to it as you don’t want it to sit idle.
* You can have a nearly even distribution of data across all machines.

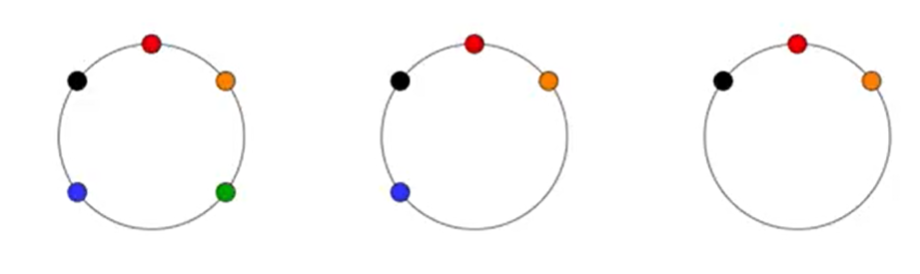
The idea is to take the number of nodes in the system out of the equation while calculating the hash. Now hashes for data and nodes will all be independently calculated and adding or removing nodes won’t change these hashes.

Now instead of assigning the data to these nodes in a sequential manner, we will plot these nodes, or rather their hashes, on the number line, and each node will be responsible for the range between its position and the position of the first node to its right. You could also define the range each node is responsible for as the range between the node itself and the first node to its left, but in this article, we will consider the range each node is responsible for as the range between the node itself and the first node to its right.

When the data comes in for storage, we will calculate its hash to determine its place on the number line. Based on which node’s range it falls in, we will map the data to that node.

If you want to remove a node from this system, the range of the previous machine on the number line will be extended, and all the data from the removed node will be mapped to the previous node in the number line. This resolves the issue of data transfer as only the data from one machine will need to be mapped to another machine.

But there is still a problem with this system. After removing a node, the ranges of certain machines might not be balanced. Try to visualize the number line as a ring as follows:



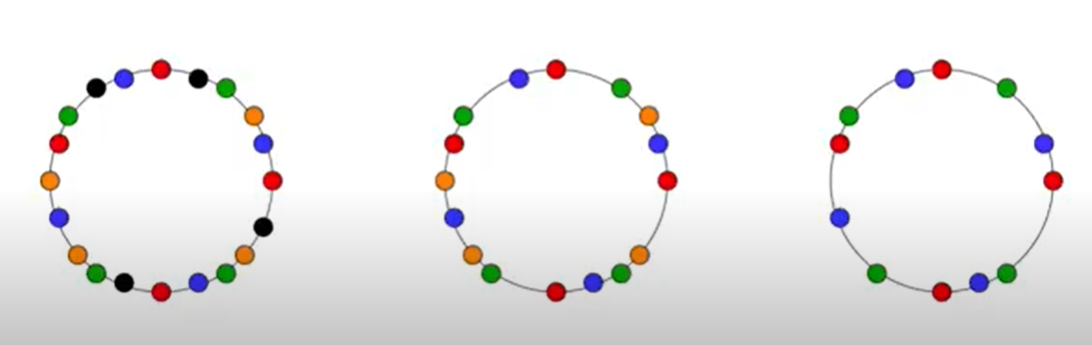
The above image starts with five nodes, and then we remove the green node. All of the green node's data will be remapped to the first node counter-clockwise to the green node (which is the orange node). Then we remove the blue node, and all of the blue node's data also gets mapped to the orange node. You can see the distribution become extremely skewed as the orange node is handling more than half the system's load. So even though the data transfer issue is pretty much resolved, there is still the problem of uneven data distribution.

An ideal solution to this would be if the data from the removed node is shared between various nodes. But how do we achieve this? There is still one improvement we can implement here.

The idea is to assign the same machine to multiple hashes and map each of these hashes on the number line. This can be achieved by either using multiple hash functions or by assigning multiple identifiers to the same node, and then calculating the hashes for all of the instanced. We can then map them on the number line, or what we can now refer to as a consistent hashing ring, essentially representing the same node on the ring multiple times.

Again each representation of the node handles the data lying between itself and the first node to its right (clockwise). Only, now the data handled by a single node will be distributed across the ring. So when a node needs to be removed, all its identifiers will be removed and the data will be mapped to the nodes falling to the right of each identifier, as shown in the following diagram, making the distribution much more even.

The reason why it gets even is because each identifier of machine is hashed separately, with the hash function, it is very likely that they would be arranged in a random fashion like Blue\_4 - Red\_1 - Blue\_2 - Green\_3 - Red\_4 - Orange\_1, which makes sure that when one machine is removed, it's data us spread across multiple other machines.



Similarly, when a new node is added, its identifiers will also be mapped across the ring, picking up data from various nodes, maintaining even distribution across the ring.

We could also add the instances of each machine different number of times, say Blue machine has 10 instances, and Green machine has 4 instances if Blue machine has more hardware capacity.

Where is it Used?

Now we just saw how we could use consistent hashing while building a caching system. There are a lot of systems out there that use consistent hashing for improving their performance. For example, Cassandra, a distributed NoSQL Columnar DB that deals with huge traffic uses consistent hashing to distribute its data. Amazon’s Dynamo DB is another such example. It is a managed distributed DB which uses consistent hashing for distributing its data. Similarly, Couchbase is another NoSQL DB (a document DB) which uses consistent hashing to distribute its data across various instances.