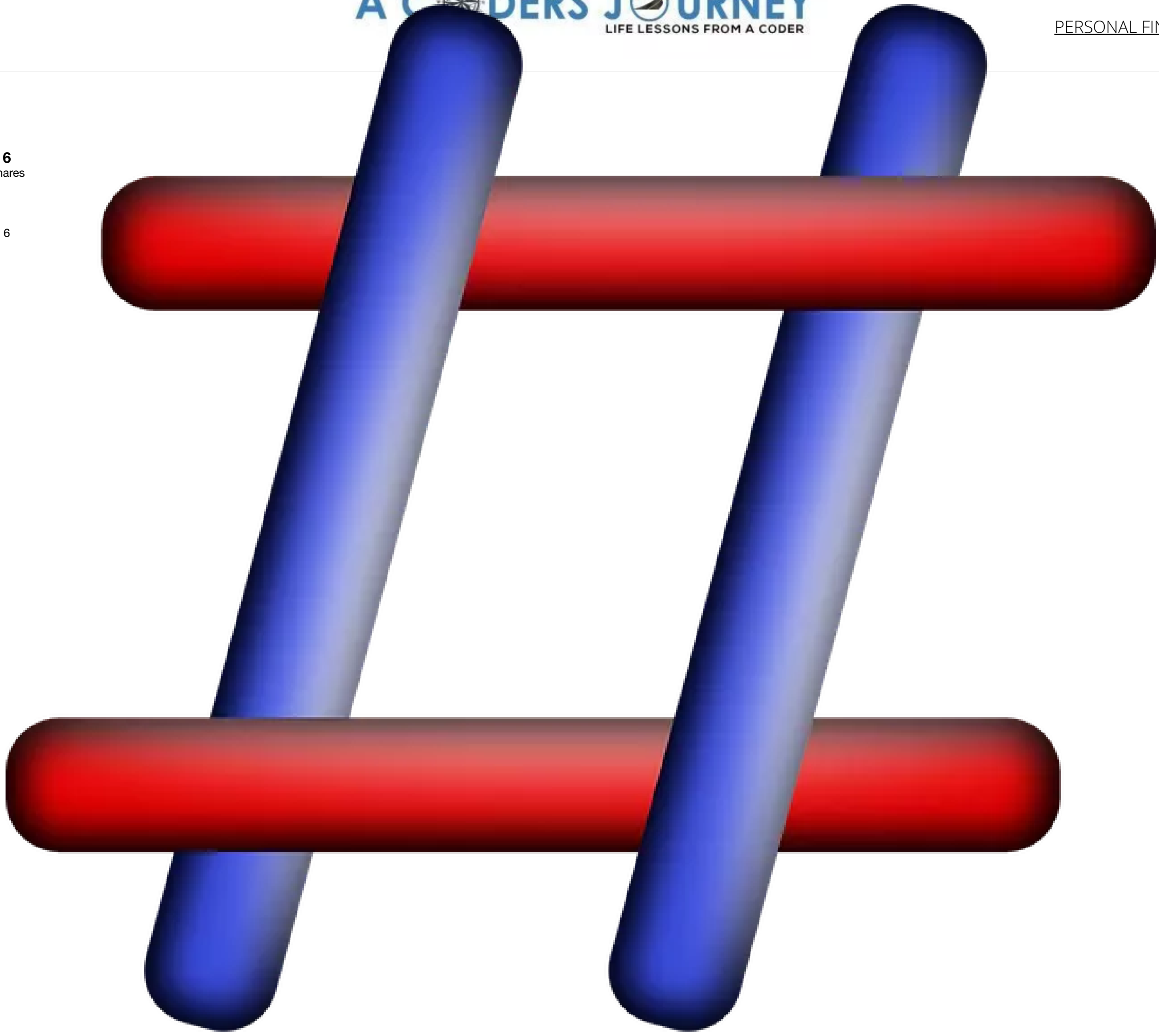
6
Shares

6


Consistent Hashing

System Design Interview Concepts – Consistent Hashing

October 9, 2017 // by Deb Haldar (https://www.acodersjourney.com/author/debohaldaryahoo-com/)

Consistent hashing is one of the techniques used to bake in scalability into the storage architecture of your system from grounds up.

In a distributed system, consistent hashing helps in solving the following scenarios:

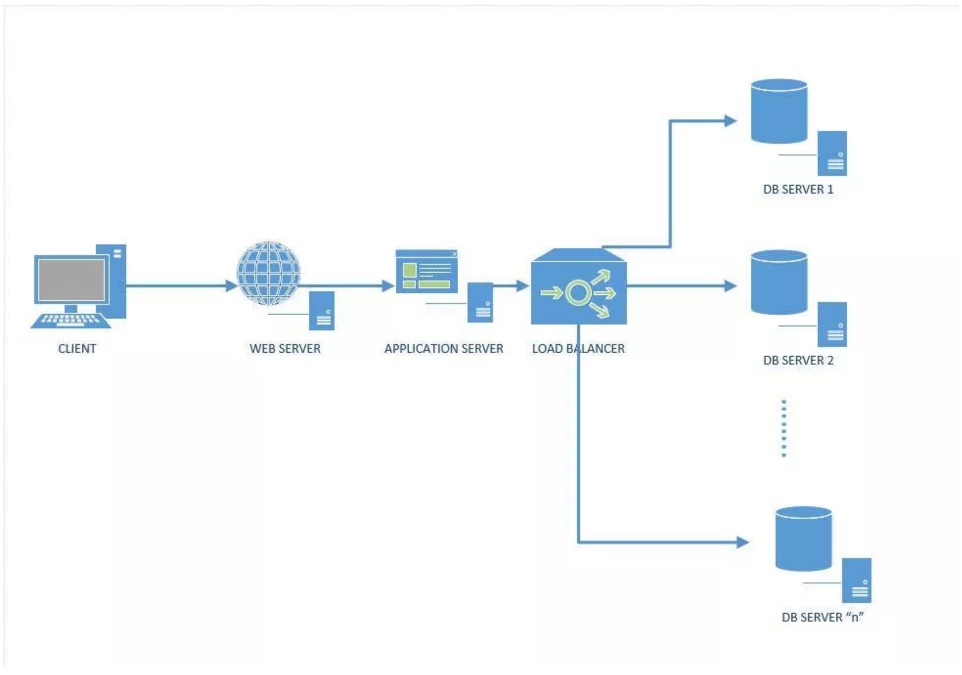
1. To provide elastic scaling (a term used to describe dynamic adding/removing of servers based on usage load) for cache servers.
2. Scale out a set of storage nodes like NoSQL databases.

It is a very useful concept that frequently comes up in System Design Interviews. You might need to apply the concept while designing the backend of a system to alleviate bottlenecks . You could also be directly asked to design and implement a consistent hashing algorithm. In this article, we'll look at:

- Why do we need Consistent Hashing ?

Why do we need Consistent Hashing ?

Imagine that you want to create a scalable database backend with "n" database servers for your web application as depicted by the diagram below. For our simple example, we'll assume that we're just storing a ***key:value*** pair like ***"Country:Canada"*** in the DBs.



(https://i2.wp.com/www.acodersjourney.com/wp-content/uploads/2017/08/Consistent-Hashing-Load-Balancing-Database-Servers.jpg)

Figure 1: A distributed system with a cluster of database servers

Our goal is to design a database storage system such that:

1. **We should be able to distribute the incoming queries uniformly among the set of "n" database servers**
2. **We should be able to dynamically add or remove a database server**
3. **When we add/remove a database server, we need to move the minimal amount of data between the servers**

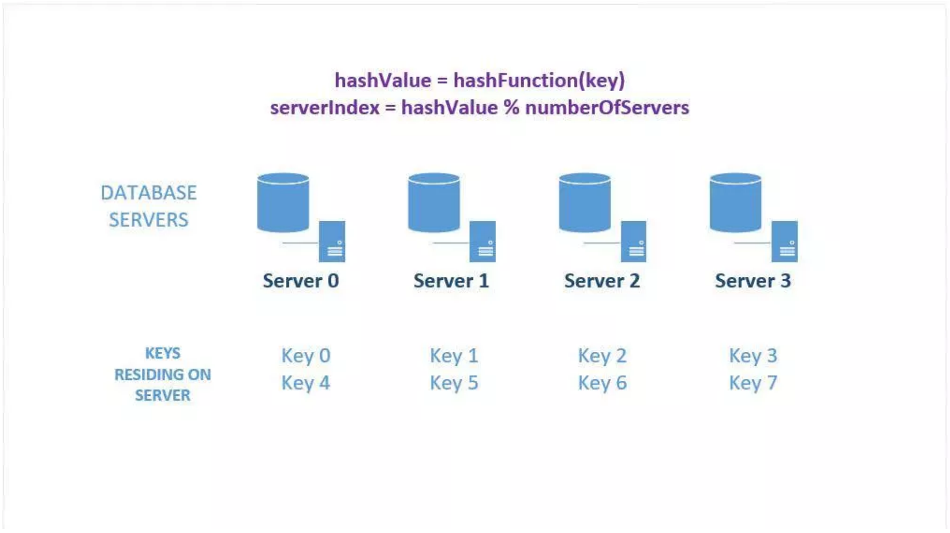
So essentially we need to send each piece of incoming query to a specific server. A simple approach is as follows:

1. Generate a hash of the key from the incoming data : " **hashValue = HashFunction(Key)**"
2. Figure out the server to send the data to by taking the modulo ("%") of the hashValue using the number of current db servers, n : "**serverIndex = hashValue % n**"

Let's walk through a simple example.

- Imagine we have 4 database servers
- Imagine our hashFunction returns a value from 0 to 7
- We'll assume that "key0" when passed through our hashFunction, generates a hashvalue or 0, "key1" generates 1 and so on.
- The serverIndex for "key0" is 0, "key1" is 1 and so on.

The situation assuming that the key data is unfirmly distributed, is depicted in the image below. We receive 8 pieces of data and our hashing algorithm distributes it evenly across our four database servers.



(https://i0.wp.com/www.acodersjourney.com/wp-content/uploads/2017/08/Data-Sharding-Based-on-Modulo-number-of-servers.jpg)

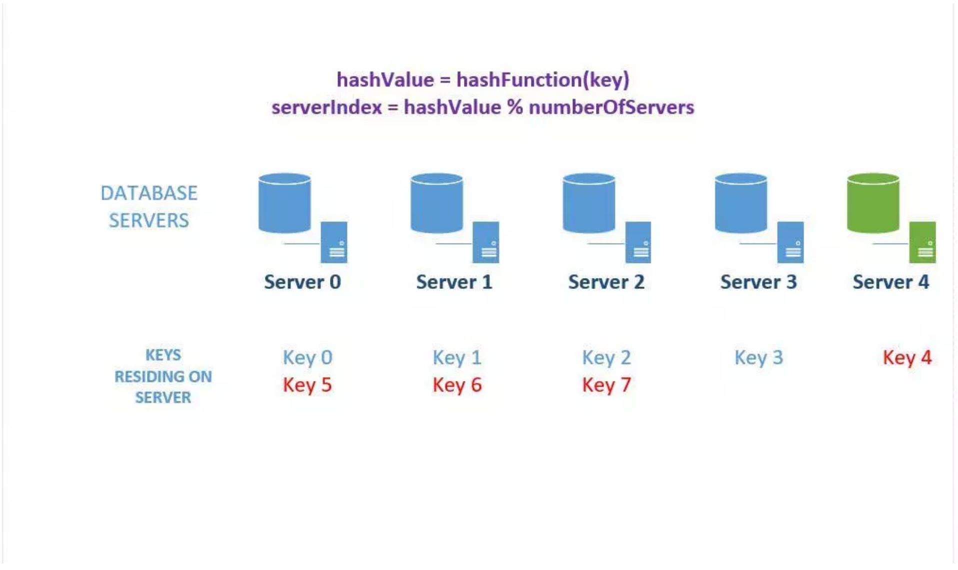
Figure 2: Sharding/ Distributing data across several database servers

Problem solved, right ? Not quite – **there's two major drawbacks with this approach, namely, Horizontal Scalability and Non-Uniform data distribution across servers.**

Horizontal Scalability

This scheme is not horizontally scalable. If we add or remove servers from the set, all our existing mappings are broken. This is because the value of "n" in our function that calculates the serverIndex changes. The result is that all existing data needs to be remapped and migrated to different servers. This might be a herculean task because it'll either require a scheduled system downtime to update mappings or creating read replicas of the existing system which can service queries during the migration. In other words, a lot of pain and expenditure.

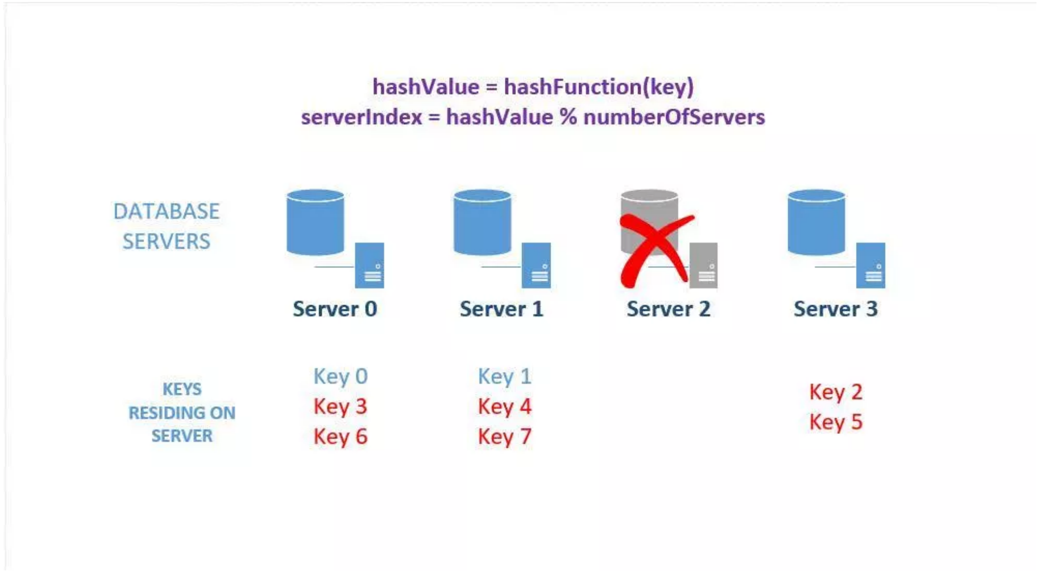
Here's a quick illustration of what happens when we add another server (server 5) to the mix. Please refer back to figure 1 for the original key distribution. Notice that we'll need to update 3 out of the original 4 servers – i.e. 75% of servers needs to be updated!



(https://i2.wp.com/www.acodersjourney.com/wp-content/uploads/2017/08/Data-Sharding-Server-Added.jpg)

Figure 3: Effect of adding a database server to the cluster

The effect is more drastic when a server goes down as depicted below. In this case, we'll need to update ALL servers, i.e., 100% of servers needs to be updated !



(https://i1.wp.com/www.acodersjourney.com/wp-content/uploads/2017/08/Data-Sharding-Server-removed.jpg)

Figure 4: Effect of removing a server from the database cluster

Data Distribution – Avoiding "Data Hot Spots" in Cluster

We cannot expect uniform distribution of data coming in all the time. There may be many more keys whose hashValue maps to server number 3 than any other servers , in which case server number 3 will become a hotspot for queries.

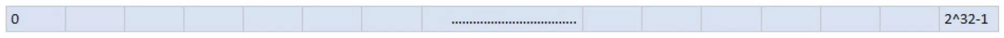
Consistent hashing allows up to solve both these problems. Read on to find out how !

How does Consistent Hashing Work ?

Consistent hashing facilitates the distribution of data across a set of nodes in such a way that minimizes the re-mapping/ reorganization of data when nodes are added or removed. Here's how it works:

1. Creating the Hash Key Space: Consider we have a hash function that generates integer hash values in the range [0, 2^32-1)

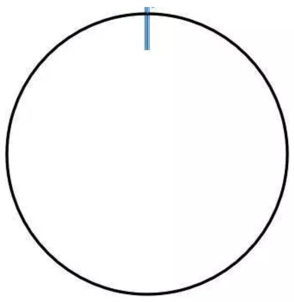
We can represent this as an array of integers with 2^32 -1 slots. We'll call the first slot x0 and the last slot xn – 1



(https://i1.wp.com/www.acodersjourney.com/wp-content/uploads/2017/08/Key-Space.jpg)

Figure 5: A Hash Key Space

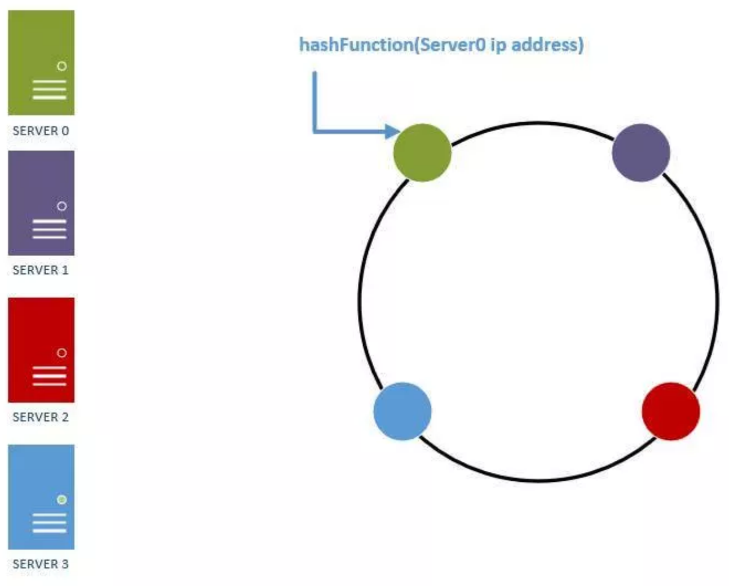
2. Representing the hashSpace as a Ring: Imagine that these integers generated in step # 2 is placed on a ring such that the last value wraps around.



(https://i1.wp.com/www.acodersjourney.com/wp-content/uploads/2017/08/HashRing.jpg)

Figure 6: Visualizing the hash key space as a Ring

3. **Placing DB Servers in Key Space (HashRing):** We're given a list of database servers to start with. Using the hash function, we map each db server to a specific place on the ring. For example, if we have 4 servers, we can use a hash of their IP addressed to map them to different integers using the hash function. This simulates placing the four servers into a different place on the ring as shown below.



(https://i1.wp.com/www.acodersjourney.com/wp-content/uploads/2017/08/Placing-DB-Servers-on-Hash-Ring.jpg)

Figure 7: Placing database servers on a hash ring

4. **Determining Placement of Keys on Servers:** To find which database server an incoming key resides on (either to insert it or query for it), we do the following:

- Run the key through the same hash function we used to determine db server placement on the ring.
- After hashing the key, we'll get an integer value which will be contained in the hash space, i.e., it can be mapped to some postion in the hash ring. There can be two cases:

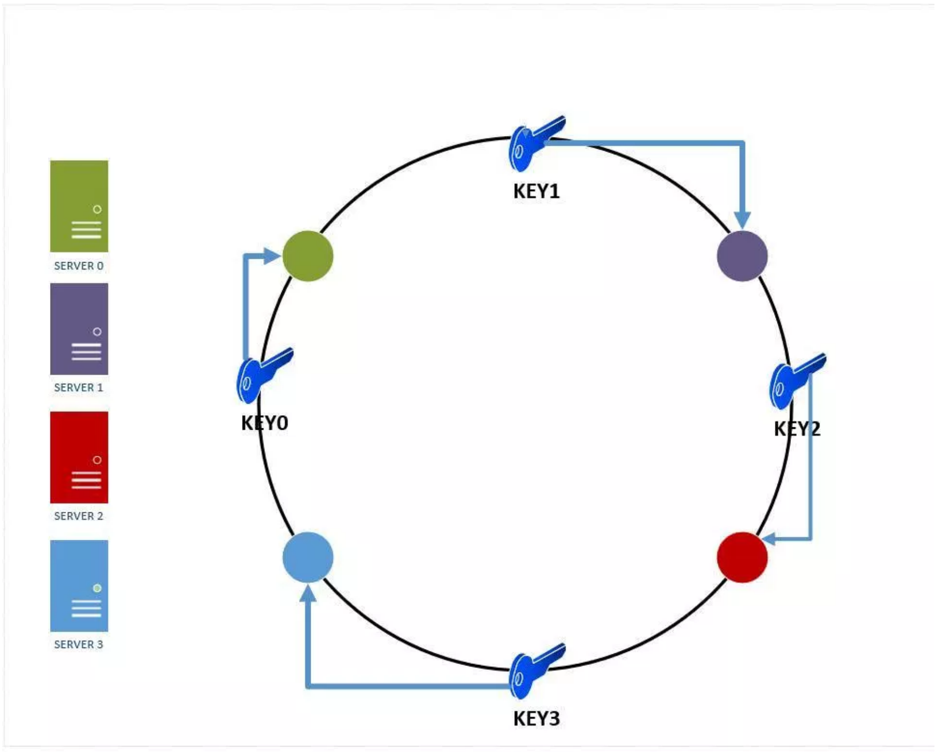
    1. The hash value maps to a place on the ring which does not have a db server. In this case, we travel clockwise on the ring from the point where the key mapped to untill we find the first db server. Once we find the first db server travelling clockwise on the ring, we insert the key there. The same logic would apply while trying to find a key in the ring.
    2. The hash value of the key maps directly onto the same hash vale of a db server – in which case we place it on that server.

2. The hash value of the key maps directly onto the same hash vale of a db server – in which case we place it on that server.

Example: Assume we have 4 incoming keys : key0, key1, key2, key3 and none of them directly maps to the hash value of any of the 4 servers on our hash ring. So we travel clockwise from the point these keys maps to in our ring till we find the first db server and insert the key there. This is depicted in Figure 7 below.

6
Shares

6

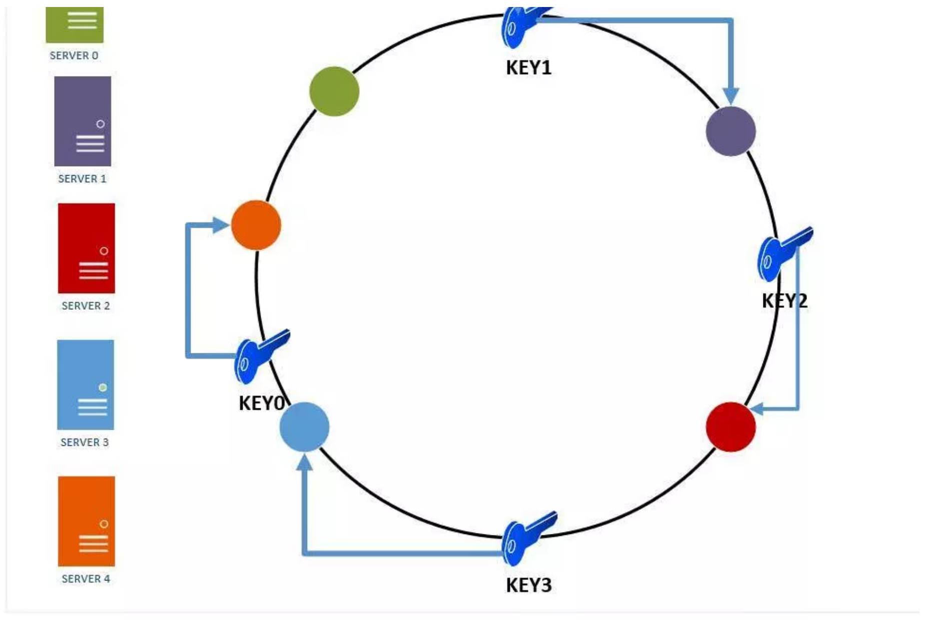


(https://i2.wp.com/www.acodersjourney.com/wp-content/uploads/2017/08/Key-Placement-on-Hash-Ring.jpg)

Figure 8: Key placements on database servers in a hash ring

5. **Adding a server to the Ring:** If we add another server to the hash Ring, server 4, we'll need to remap the keys. However, ONLY the keys that reside between server 3 and server 0 needs to be remapped to server 4. **On an average , we'll need to remap only k/n keys , where k is the number of keys and n is the number of servers.** This is in sharp contrast to our modulo based placement approach where we needed to remap nearly all the keys.

The figure below shows the effect of inserting a new server4 – since server 4 now resides between key0 and server0, key0 will be remapped from server0 to server4.



6
Shares

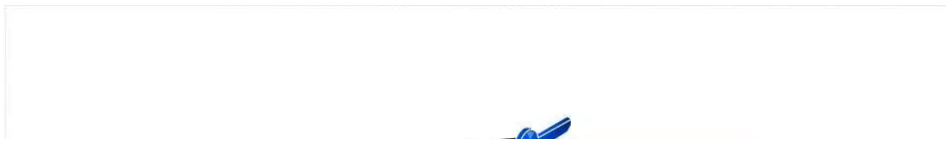
6

(https://i2.wp.com/www.acodersjourney.com/wp-content/uploads/2017/08/Consistent-Hashing-Adding-Servers.jpg)

Figure 9: Effect of adding a server to the hash ring

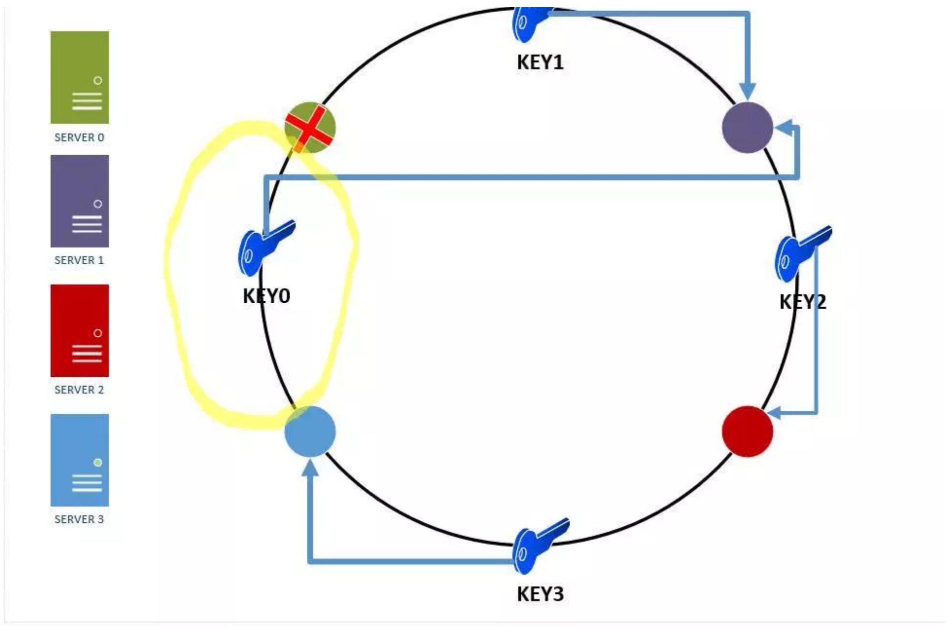
6. **Removing a server from the ring:** A server might go down in production and our consistent hashing scheme ensures that it has minimal effect on the number of keys and servers affected.

As we can see in the figure below, if server0 goes down, only the keys in between server3 and server 0 will need to be remapped to server 1(the area is circled in yellow). The rest of the keys are unaffected .



6
Shares

6



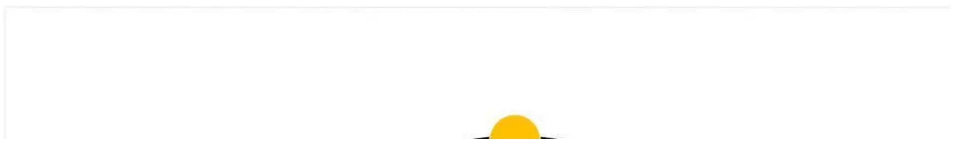
(https://i1.wp.com/www.acodersjourney.com/wp-content/uploads/2017/08/Consistent-Hashing-Removing-Servers.jpg)

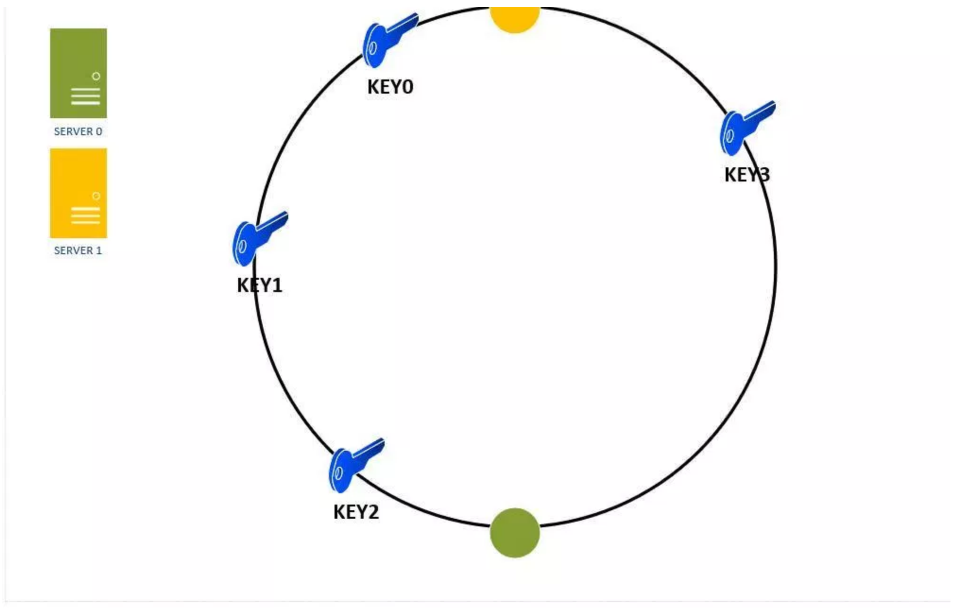
Figure 10: Effect of removing a server from the hash ring

At this point, **consistent hashing** has successfully solved the **horizontal scalability problem** by ensuring that every time we scale up or down, we DO NOT have to re-arrange all the keys or touch all the database servers !

But what about the distribution of data across the various database servers? We can run into a situation where our server distribution across the hash ring is non uniform , i.e., the size of partitions each server is responsible for is not the same. But you might ask how will that happen ? Well, imagine that we started off with 3 servers (server0, server1, server2) that were more or less evenly distributed across the ring. If one of the servers fail, then the load seen by the server immediately following the failed server will be higher. This is assuming all the data that comes in has a uniform key distribution. In reality , the issue is more complicated because data does not have uniform distribution in most cases. So these two things coupled together can lead to a situation like the one shown below. Here, server0 is seeing a very high load because :

1. Data was non-uniformly distributed to start with – so server2 was having a lot of hot spots
2. Server2 eventually fails and had to be removed from the hash ring. (note that server 0 now gets all of server2's keys)





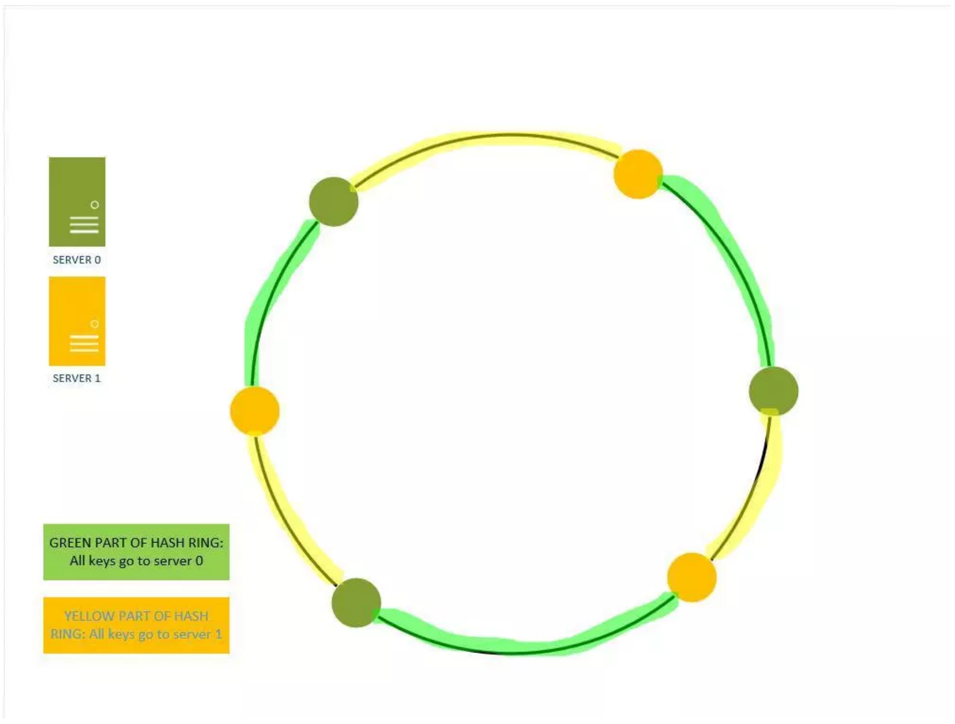
(https://i1.wp.com/www.acodersjourney.com/wp-content/uploads/2017/08/Consistent-Hashing-Non-Uniform-Distribution-No-Replicas.jpg)

Figure 11: Keys can be non-uniformly distributed across servers in a hash ring

So how do we solve this ?

It turns out that there is a pretty standard solution to the problem. It involves the introduction of a number of replicas or virtual nodes for each server across the ring. For example,

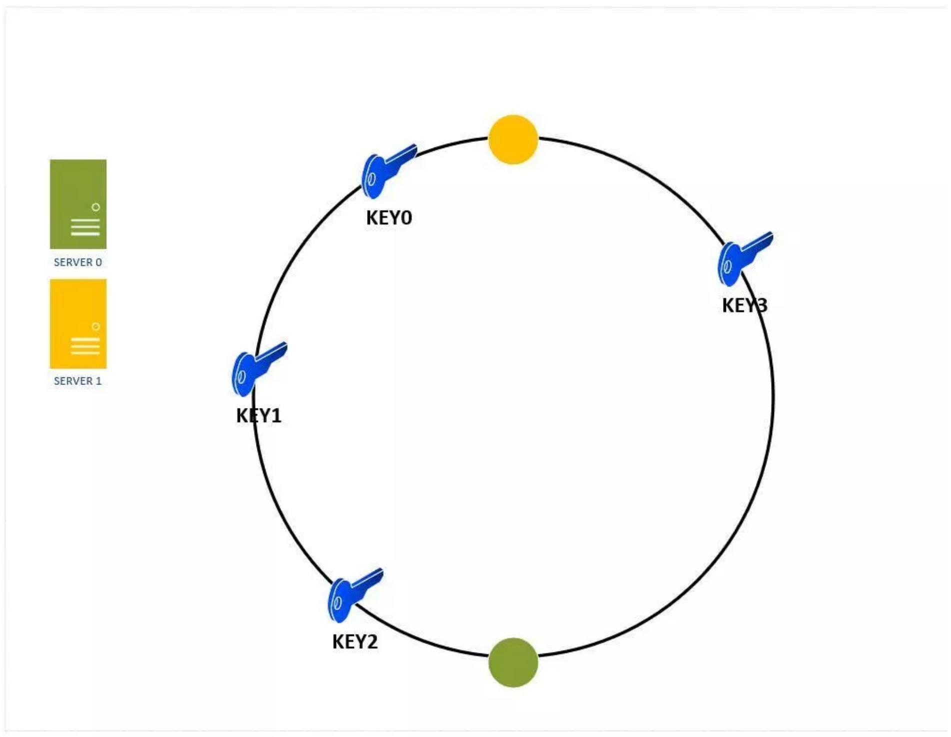
Server 0 might have two replicas placed at different points across the ring.


SERVER 0
SERVER 1
GREEN PART OF HASH RING: All keys go to server 0
YELLOW PART OF HASH RING: All keys go to server 1

(https://i2.wp.com/www.acodersjourney.com/wp-content/uploads/2017/08/Consistent-Hashing-Virtual-Nodes.jpg)

Figure 12: Using Virtual Nodes to assign increase the key space covered by each server

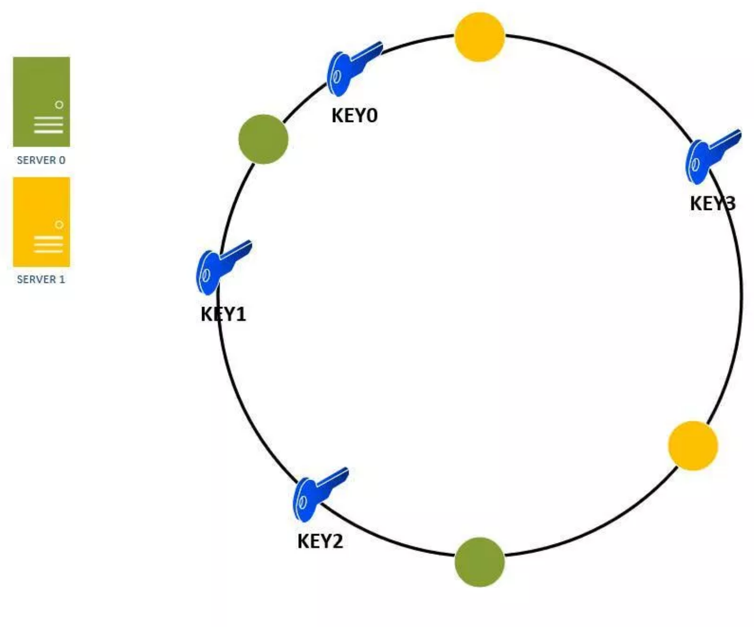
But how does using replicas make the key distribution more uniform ? Here's a visual example – Figure 13 shows the key distribution with two serevers in the hash ring WITHOUT replicas. We can observe that server 0 is handling 100% of the keys.



(https://i1.wp.com/www.acodersjourney.com/wp-content/uploads/2017/08/Consistent-Hashing-Non-Uniform-Distribution-No-Replicas.jpg)

Figure 13: Non-uniform key distribution in absence of replication of nodes in a hash ring

If we introduce one more replica of each server on the ring , then the key distribution looks like the one in figure 14. Now server0 is responsible for 50% (2 out of 4) keys and server 1 is responsible for the other 50% of the keys.



(https://i2.wp.com/www.acodersjourney.com/wp-content/uploads/2017/08/Consistent-Hashing-Replication-Alleviates-Hotspots.jpg)

Figure 14: Using virtual nodes/ replication to create better key distribution in a hash ring

As the number of replicas or virtual nodes in the hash ring increase, the key distribution becomes more and more uniform. In real systems, the number of virtual nodes / replicas is very large (>100) .

At this point, **Consistent Hashing** has successfully solved the problem of **non-uniform data distribution** (hot spots) across our database server cluster.

Key things to remember about Consistent Hashing for System Design Interviews

SCENARIOS WHERE TO USE CONSISTENT HASHING

1. You have a cluster of databases and you need to elastically scale them up or down based on traffic load. For example, add more servers during Christmas to handle the the extra traffic.
2. You have a set of cache servers that need to elastically scale up or down based on traffic load.

BENEFITS OF CONSISTENT HASHING:

1. Enables Elastic Scaling of cluster of database/cache servers
2. Facilitates Replication and partitioning of data across servers
3. Partitioning of data enables uniform distribution which relieves hot spots
4. Points a-c enables higher availability of the system as a whole.

Implementation Consistent Hashing

Please note that this is for purely illustrative purposes only. There are no guarantees for robustness or stability if used in production code.

There are three key pieces we need to implement:

1. A Hash table like data structure which can simulate the key space or the hash Ring. In our case, we'll use a SortedDictionary in C#
2. A hash function that can generate a integer value for the servers ip address and incoming keys we need to map to the hash ring
3. The server object themselves.

First we define a server class which basically encapsulates an ip address and represent a physical server.

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsistentHashing
{
    class Server
    {
        public String ipAddress;

        public Server(String ipAddress)
        {
            this.ipAddress = ipAddress;
        }
    }
}
```

Next we define the hash function which will return an integer value for server ips and the keys.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
/*
 * This code is taken from the stackoverflow article:
 * https://stackoverflow.com/questions/12272296/32-bit-fast-uniform-hash-function-use-md5-sha1-and-cut-off-4-bytes
 */
namespace ConsistentHashing
{
    public static class FNVHash
    {
     public static uint To32BitFnv1aHash(string toHash, bool separateUpperByte = false)
```

```
        public static uint To32BitFnv1aHash(string toHash, bool separateUpperByte = false)
        {
            IEnumerable<byte> bytesToHash;

            if (separateUpperByte)
                bytesToHash = toHash.ToCharArray()
                    .Select(c => new[] { (byte)((c - (byte)c) >> 8), (byte)c })
                    .SelectMany(c => c);
            else
                bytesToHash = toHash.ToCharArray()
                    .Select(Convert.ToByte);

            //this is the actual hash function; very simple
            uint hash = FnvConstants.FnvOffset32;

            foreach (var chunk in bytesToHash)
            {
                hash ^= chunk;
                hash *= FnvConstants.FnvPrime32;
            }

            return hash;
        }
    }
    public static class FnvConstants
    {
        public static readonly uint FnvPrime32 = 16777619;
        public static readonly ulong FnvPrime64 = 1099511628211;
        public static readonly uint FnvOffset32 = 2166136261;
        public static readonly ulong FnvOffset64 = 14695981039346656037;
    }
}
```

Finally, we define the consistent hash class which enacpsulates the logic for :

1. Creating the hash ring
2. Adding a server to the hash ring
3. Removing a server from the hash ring
4. Getting the location of the server on the hash ring where a key needs to be added / retrieved from.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsistentHashing
{
    class ConsistentHash
    {
        private SortedDictionary<uint, Server> hashRing;
        private int numberOfReplicas; // The number of virtual nodes

        public ConsistentHash(int numberOfReplicas, List<Server> servers)
        {
            this.numberOfReplicas = numberOfReplicas;

            hashRing = new SortedDictionary<uint, Server>();

            if(servers != null)
            foreach(Server s in servers)
            {
                    this.addServerToHashRing(s);
            }
        }

        public void addServerToHashRing(Server server)
        {
            for(int i=0; i < numberOfReplicas; i++)
            {
                //Fuse the server ip with the replica number
                string serverIdentity = String.Concat(server.ipAddress, ":", i);
                //Get the hash key of the server
                uint hashKey = FNVHash.To32BitFnv1aHash(serverIdentity);
                //Insert the server at the hashkey in the Sorted Dictionary
                this.hashRing.Add(hashKey, server);
            }
            }
        }

        public void removeServerFromHashRing(Server server)
        {
            for (int i = 0; i < numberOfReplicas; i++)
            {
                //Fuse the server ip with the replica number
                string serverIdentity = String.Concat(server.ipAddress, ":", i);
                //Get the hash key of the server
                uint hashKey = FNVHash.To32BitFnv1aHash(serverIdentity);
                //Insert the server at the hashkey in the Sorted Dictionary
                this.hashRing.Remove(hashKey);
            }
        }

        // Get the Physical server where a key is mapped to
        public Server GetServerForKey(String key)
        {
            Server serverHoldingKey;

            if(this.hashRing.Count==0)
            {
                return null;
            }

            // Get the hash for the key
            uint hashKey = FNVHash.To32BitFnv1aHash(key);

            if(this.hashRing.ContainsKey(hashKey))
            {

                serverHoldingKey = this.hashRing[hashKey];
            }
            else
            {
                uint[] sortedKeys = this.hashRing.Keys.ToArray();

                //Find the first server key greater than  the hashkey
                uint firstServerKey = sortedKeys.FirstOrDefault(x => x >= hashKey);

                // Get the Server at that Hashkey
                serverHoldingKey = this.hashRing[firstServerKey];
            }

            return serverHoldingKey;
        }

    }
}
```

Finally, here's a test program which exercises the functionality of the above code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Security.Cryptography;

namespace ConsistentHashing
{
    class Program
    {
        static void Main(string[] args)
        {
            List<Server> rackServers = new List<Server>();
            rackServers.Add(new Server("10.0.0.1"));
            rackServers.Add(new Server("10.0.0.2"));

            int numberOfReplicas = 1;

            ConsistentHash serverDistributor = new ConsistentHash(numberOfReplicas, rackServers);

            //add a new server to the mix
            Server newServer = new Server("10.0.0.3");
            serverDistributor.addServerToHashRing(newServer);

            //Assume you have a key "key0"
            Server serverForKey = serverDistributor.GetServerForKey("key0");
            Console.WriteLine("Server: " + serverForKey.ipAddress + " holds key: Key0");

            // Now remove a server
            serverDistributor.removeServerFromHashRing(newServer);
            // Now check on which server "key0" landed up
            serverForKey = serverDistributor.GetServerForKey("key0");
            Console.WriteLine("Server: " + serverForKey.ipAddress + " holds key: Key0");

        }
    }
}
```

6
Shares

OUTPUT:

6

```
Server: 10.0.0.3 holds key: Key0
Server: 10.0.0.2 holds key: Key0
```

Consistent Hashing in Action in Production Systems

There are a number of live systems which use consistent hashing including:

- Couchbase (https://en.wikipedia.org/wiki/Couchbase) automated data partitioning
- Partitioning component of Amazon's storage system Dynamo (https://en.wikipedia.org/wiki/Dynamo_(storage_system))
- Data partitioning in Apache Cassandra (https://en.wikipedia.org/wiki/Apache_Cassandra)
- Riak (https://en.wikipedia.org/wiki/Riak), a distributed key-value database
- Akamai (https://en.wikipedia.org/wiki/Akamai_Technologies) Content Delivery Network
- Discord (https://en.wikipedia.org/wiki/Discord_(software)) chat application

Further Reading On Consistent Hashing

1. Tom White's (http://www.tom-e-white.com/2007/11/consistent-hashing.html) article on Consistent Hashing is the one i used to initially learn about this technique. The C# implementation in this article is loosely based on his java implementation.

2. Tim Berglund's Distributed System in One Lesson (https://www.safaribooksonline.com/library/view/distributed-systems-in/9781491924914/video215269.html?autoStart=True) is a fantastic resource to learn about read replication, sharding and consistent hashing. Unfortunately, you'll need a safari membership for this.

3. David Karger and Eric Lehman's (http://www.akamai.com/us/en/multimedia/documents/technical-publication/consistent-hashing-and-random-trees-distributed-caching-protocols-for-relieving-hot-spots-on-the-world-wide-web-technical-publication.pdf) original paper on Consistent Hashing

4. David Karger and Alex Sherman's paper (http://www8.org/w8-papers/2a-webserver/caching/paper2.html) on Web Caching with Consistent Hashing

If you have any feedback, please add it to the comment section below. And if you enjoyed the article, please share it on your favorite social media platform 🙂

« Top 20 C++ multithreading mistakes and how to avoid them (https://www.acodersjourney.com/top-20-cplusplus-multithreading-mistakes/)

35 things I learnt at Game Developer Conference (GDC) 2018 » (https://www.acodersjourney.com/gdc-2018-35-key-insights/)

10 Comments A Coders Journey 1 Login