Consistent Hashing

- Consistent hashing has become very common in the last decade or so due to increasing data sizes, resulting in data being distributed across multiple computers, and also due to distributed compute platforms become more widely used.

- Imagine an in-memory cache that is hosted on a computer.
  - It’s size is such that it all fits in the RAM of the computer.

- Consistent hashing has become very common in the last decade or so due to increasing data sizes, resulting in data being distributed across multiple computers, and also due to distributed compute platforms become more widely used.

- Imagine an in-memory cache that is hosted on a computer.
  - It's size is such that it all fits in the RAM of the computer.

- Now, imagine the cache size needs to become 5X:
  - We could scale up the computer (*vertical* scaling).
    - This means, we can get a sku with a larger RAM. ← more expensive computers.

- Consistent hashing has become very common in the last decade or so due to increasing data sizes, resulting in data being distributed across multiple computers, and also due to distributed compute platforms become more widely used.

- Imagine an in-memory cache that is hosted on a computer.
  - It's size is such that it all fits in the RAM of the computer.

- Now, imagine the cache size needs to become 5X:
  - We could scale up the computer (*vertical* scaling).
  - This means, we can get a sku with a larger RAM. ← more expensive computers

  - But this solution doesn't really scale very well.
    - Imagine data size growth of 10X or 100X or more.

- Consistent hashing has become very common in the last decade or so due to increasing data sizes, resulting in data being distributed across multiple computers, and also due to distributed compute platforms become more widely used.

- Imagine an in-memory cache that is hosted on a computer.
  - It's size is such that it all fits in the RAM of the computer.

- Now, imagine the cache size needs to become 5X:
  - We could scale up the computer (*vertical* scaling).
  - This means, we can get a sku with a larger RAM. ← more expensive computers

  - But this solution doesn't really scale very well.
    - Imagine data size growth of 10X or 100X or more.
    - We can scale out (*horizontal* scaling) … this means we spread the data across multiple computers.
    - But if we scale it out, we need to know which computer to write a piece of data to (or read from).

In a cache that is spread across multiple computers, mapping a piece of data to a cache server is something that is done all the time (for reads and writes).

This cache system is known as a *distributed* cache.

Examples: memcached, Redis, Riak, Oracle Coherence. These are key-value data stores.

In a cache that is spread across multiple computers, mapping a piece of data to a cache server is something that is done all the time (for reads and writes).

This cache system is known as a *distributed* cache.

Examples: memcached, Redis, Riak, Oracle Coherence. These are key-value data stores.

Use cases:

1. Key value stores where u want to quickly access the data, given a key.
2. Front end cache to a database.
    1. Example:
        1. An online travel company could have a cache in front of a database, because typically the search traffic is far more than actual booking traffic.
        2. u can use ur own experience…how many hotels do u first look at before u actually book.
        3. Or how many items u shop at some online store before u actually purchase.

Use cases:

3. Once customer wants to purchase, final pricing call goes to the database
   - freshest data, since that is the source.
4. Note that it is possible in some cases (hotel or flight bookings) that the price changes once u go to the purchasing page (u may have experienced this).
   - This is not a bug (well, could be ☺), but its because the cache can be out of sync with the database in situations where the database gets frequent updates on price changes / availability … which it does in the travel industry (deals, etc). But perhaps not so much on a non-travel shopping site (like shopping for shoes, shirts, etc).

Distributed cache:

Now, think back about the Hash tables we had talked about.
In hash tables, we map a key to a location in the hash table.
We do this by:

Hash value modulo hash table size.

Now, think back about the Hash tables we had talked about.
In hash tables, we map a key to a location in the hash table.
We do this by:

Hash value modulo hash table size.

In a distributed cache, something similar is done.
Since we need to map a key to a cache server, we use the number of servers instead of hash table size.

So, u can think of it as:

HashValue (key) % NumServers ➔ server number

Example:
Lets say we have 10 servers: S0 thru S9 ➔ NumServers = 10

I add the following keys:

Key	hash	Server number
“university”	Hash (key) % NumServers	1
“washington”	Hash (key) % NumServers	5
“geography”	Hash (key) % NumServers	9
“oregon”	Hash (key) % NumServers	1

So, u can think of it as:

HashValue (key) % NumServers ➔ server number

Now, what happens if: ← Example slide on this coming up

1. We lose a server (went down due to some fault), or
2. We need to add a server because we are low on capacity (memory, bandwidth, etc.)

- Think of #2 to be roughly equivalent to when we have to increase the capacity of a hash table.

So, u can think of it as:

HashValue (key) % NumServers ➔ server number

Now, what happens if:

1. We lose a server (went down due to some fault), or
2. We need to add a server because we are low on capacity (memory, bandwidth, etc.)

- Think of #2 to be roughly equivalent to when we have to increase the capacity of a hash table.
- There, we had to rehash all the values and insert them into the resized hash table.
- What would happen if we didn't resize, and simply copied the existing data from old hash table to new.
  - We would look for data in the wrong place…so:
    - we would have cache misses for all the existing data.
    - But new data entered now would give us a cache hit when we come back to read it .

So, u can think of it as:

HashValue (key) % NumServers ➔ server number

Now, what happens if:

1. We lose a server (went down due to some fault), or
2. We need to add a server because we are low on capacity (memory, bandwidth, etc.)

Now, back to a distributed cache … if we add or remove a server, the NumServers part would change, resulting in a change in the server number that a key gets mapped to.

So, the impact of adding or removing a cache server is that a read of any of the existing keys will now result in a cache miss.

See example next

HashValue (key) % NumServers ➔ server number

Example:
Now, the number of servers changes.
Lets say NumServers = 9

Key	hash	Server number
“university”	Hash (key) % NumServers	1
“washington”	Hash (key) % NumServers	5
“geography”	Hash (key) % NumServers	9
“oregon”	Hash (key) % NumServers	1

Hash (key) % NumServers will now have different values.

Hash (“washington”) % NumServers may give a number different than 5.

Hence, a read of the keys would go to a different server than where it went earlier when we had 10 servers.

And the worst part is, this affects all the keys, not just the keys on the server we lost.

But removal (or addition) of a cache server is inevitable.

- This is unlike the hash table situation, where, if u knew for sure what the size of the table should be, u would never need to resize.

- Recomputing the hash of all existing entries, and then moving them around the potentially 100s of cache servers is not a practical solution.

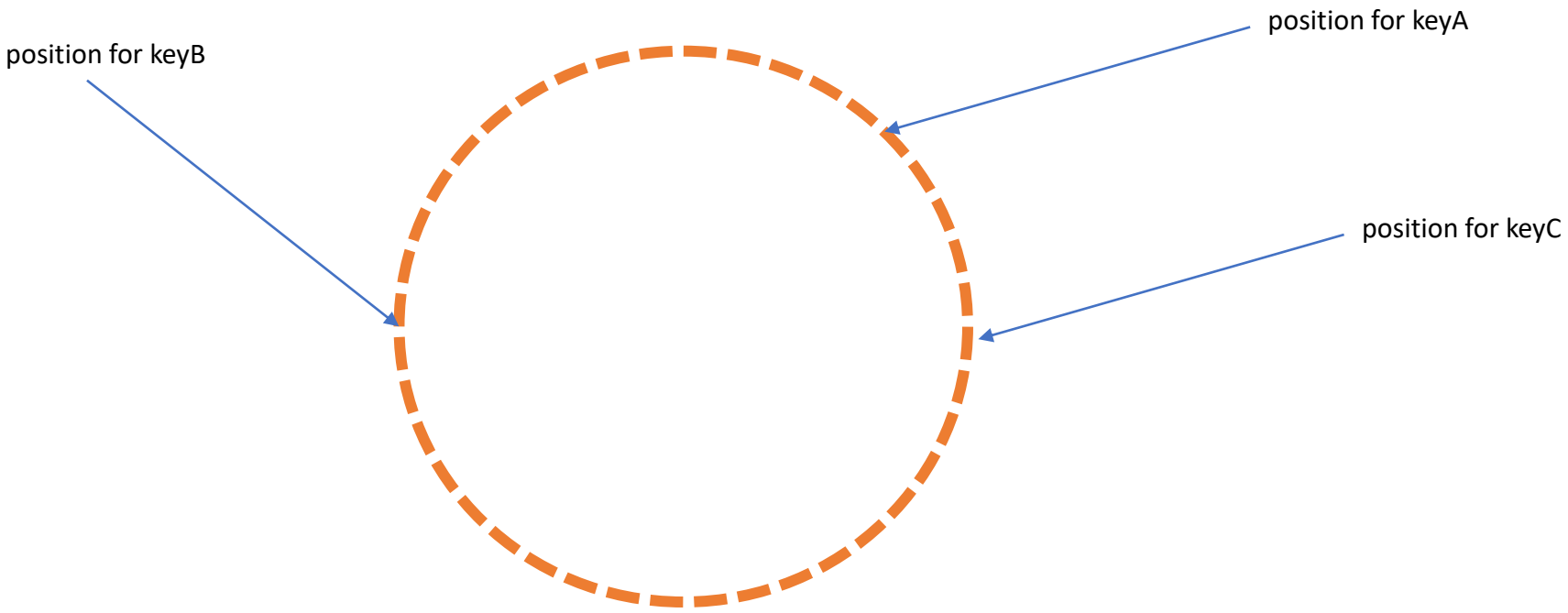
But removal (or addition) of a cache server is inevitable.

- This is unlike the hash table situation, where, if u knew for sure what the size of the table should be, u would never need to resize.

- Recomputing the hash of all existing entries, and then moving them around the potentially 100s of cache servers is not a practical solution.

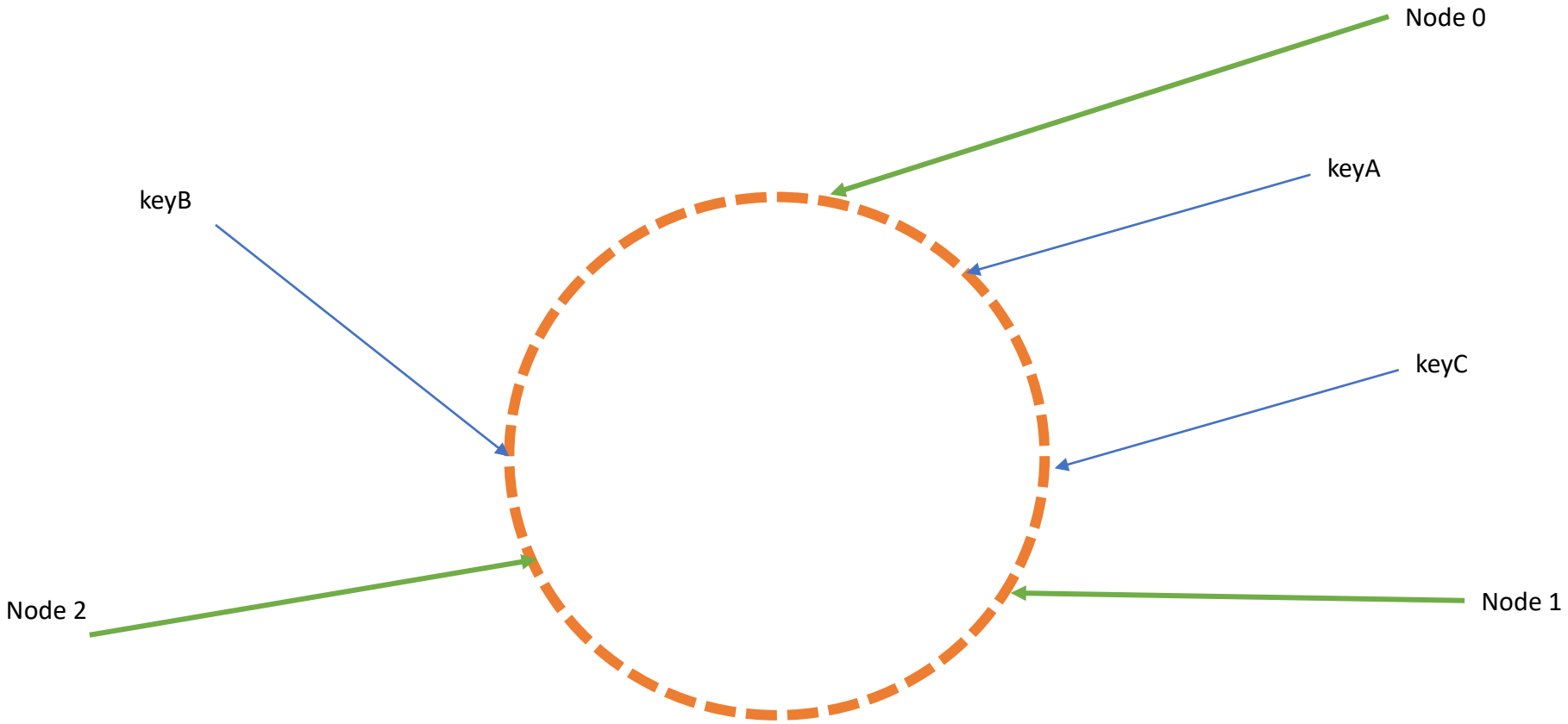
- And this is where the concept of consistent caching comes into play.
- It mitigates this problem (does not solve it).

When we talked about hashing in a hash table, we would map a key to a particular location in the hash table. Now, instead of a hash table, imagine a ring, and we will map the keys to a position on the ring (aka hash ring)



And we will also map the cache servers (nodes) to this ring.

Next: mapping the keys to cache servers (nodes)

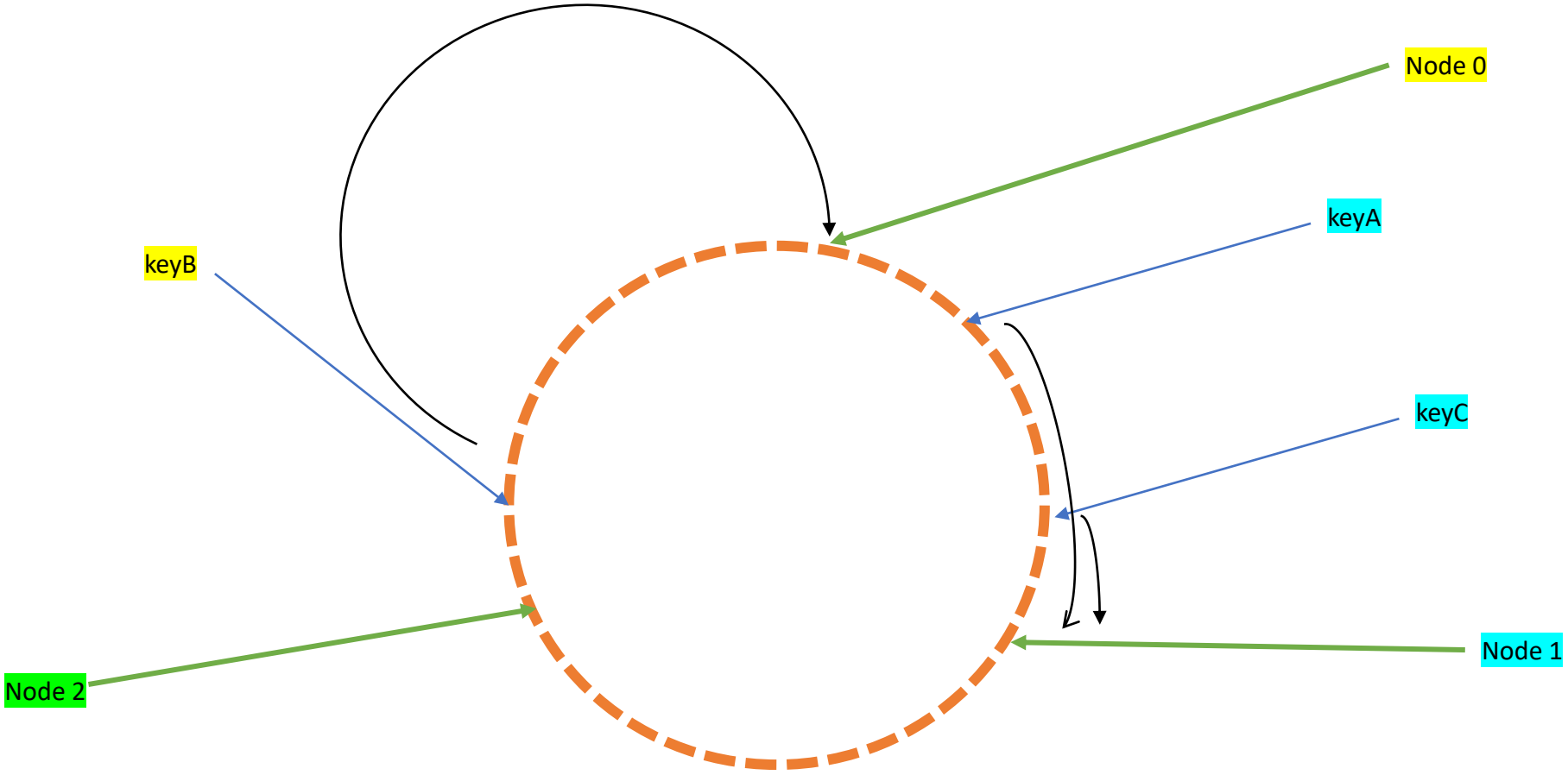


Convention:

A key belongs to the first node that is clockwise adjacent to the key

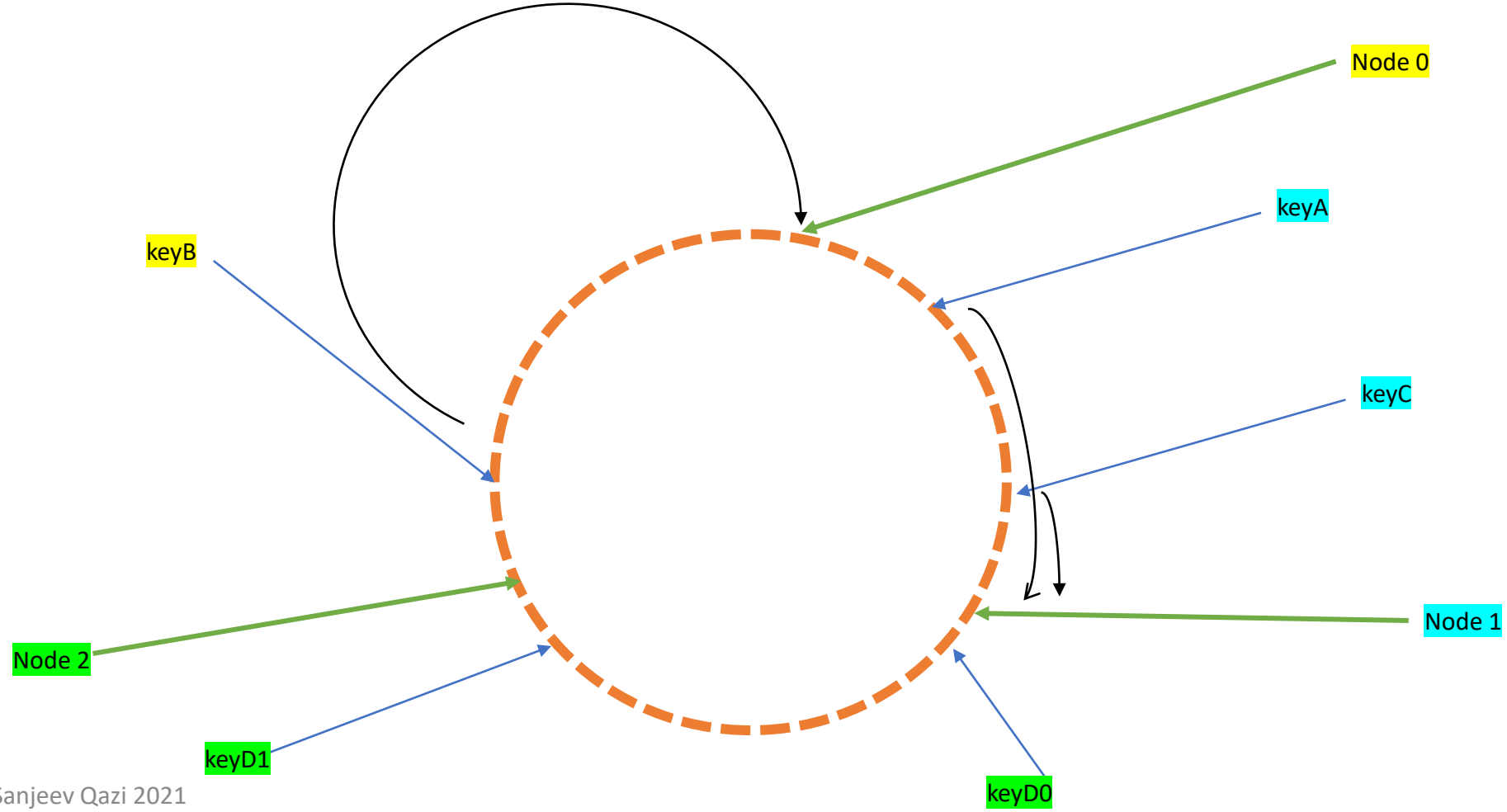
alternatively, the convention could be to pick anti-clockwise.

In this diagram, no key belongs to Node2 (just because I drew it that way ☺ … but lets assign it a key next)



Convention:

A key belongs to the first node that is clockwise adjacent to the key.



So, what we r saying is that we compute the hash value of a key, and map it to the ring.
Lets look at what it means to map to a hash ring.

Hash (Key) % 360 ➔ position on the ring. (360 represents 360 degrees of a circle).

Or

we can do modulus with some other number (no table size here, and we wont use number of servers)
Hash (Key) % SomeLargeNumber

And then map the result to one of the cache servers. ➔ We will look at how to map

Note:
Since we do not use the number of servers in the modulus, it means the value is independent of number of servers (and hence independent of “adding” or “removing” servers)

Note that the hash ring is a conceptual thing, essentially what we want is to map a key to a server, and the ring just helps visualize that we need to slide forward (clockwise in the hash ring) in the value generated to find the cache server to which the key belongs.

Note: As mentioned earlier, the convention could be to slide backwards (anti clockwise in the ring)

Mapping to servers

There are many ways to map keys to servers.

We could store the values for the cache servers in a sorted array S.

Then, to determine what server a key's value maps to, we do a binary search for the key's value in S.

Either we find that exact value (cache server's value == key's value)

or

Find cache server whose value is clockwise adjacent to key's value. ← (cache server's value > key's value)

And that's the cache server for this key.

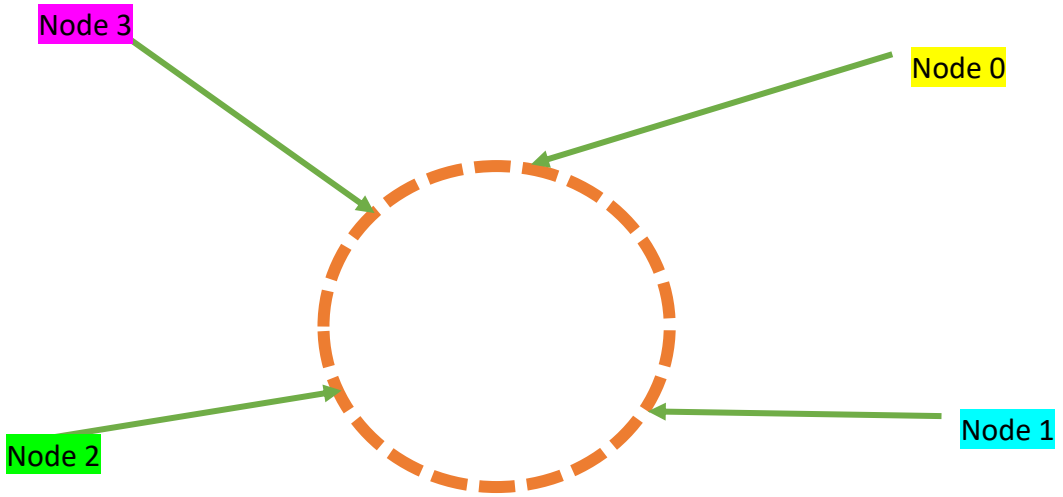
Lets look at an example next.

Mapping to servers

1. Values are just made up by me ☺, not computed
2. Key assignments next.

Cache server	Value
Node0	7
Node1	17778
Node2	77997
Node3	127017

Key	Value	Cache server
KeyA	5020	Node1
KeyB	88610	Node3
KeyC	16100	Node1
KeyD0	55330	Node2
KeyD1	61200	Node2
KeyE	142000	Node0



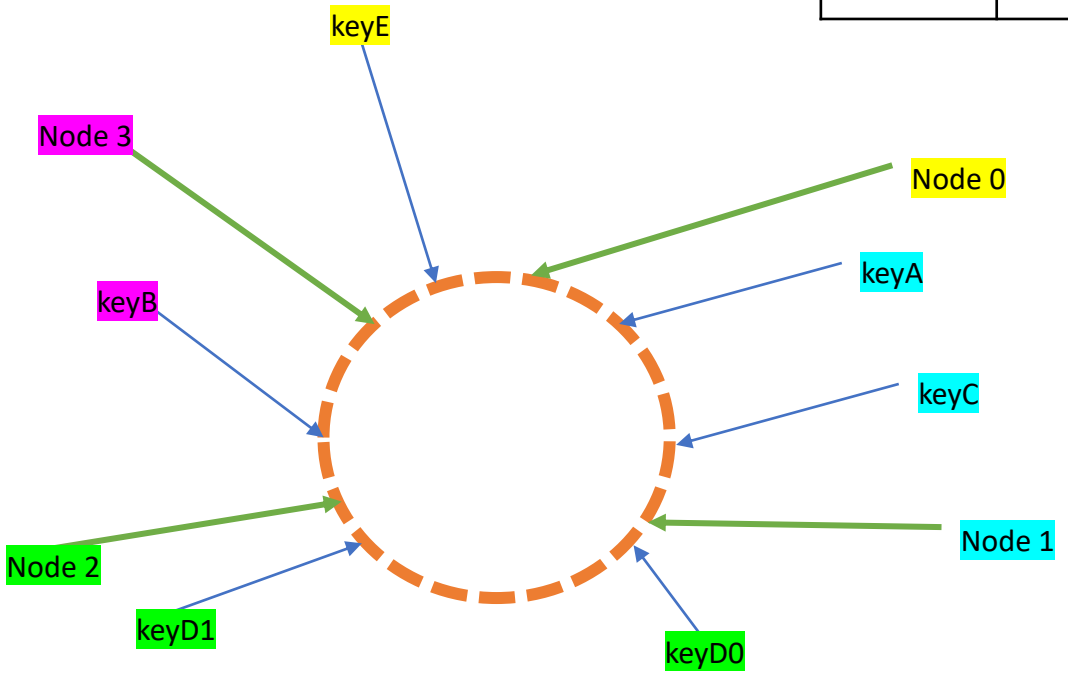
Mapping to servers

1. Values are just made up by me ☺, not computed

2. Notice how keyE "wraps around" to Node0, because no server has a value greater than keyE's value.

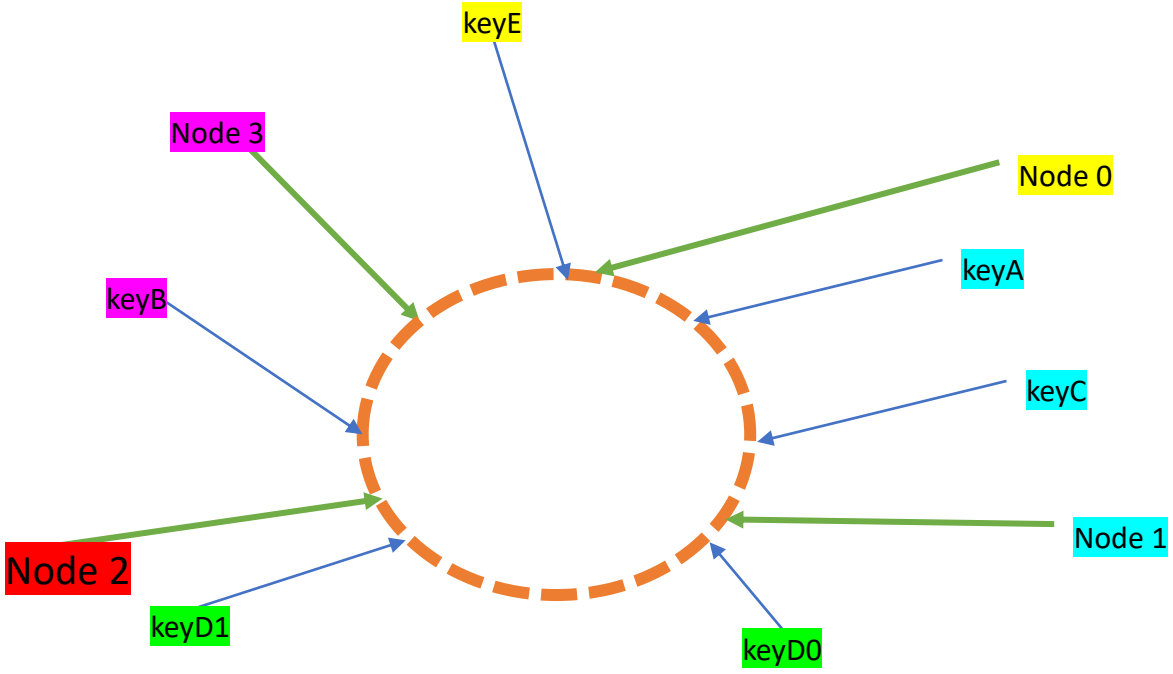
Cache server	Value
Node0	7
Node1	17778
Node2	77997
Node3	127017

Key	Value	Cache server
KeyA	5020	Node1
KeyB	88610	Node3
KeyC	16100	Node1
KeyD0	55330	Node2
KeyD1	61200	Node2
KeyE	142000	Node0



Mapping to servers

Now, what happens if some node, say, Node2 goes down.



Mapping to servers

Now, what happens if some node, say, Node2 goes down.

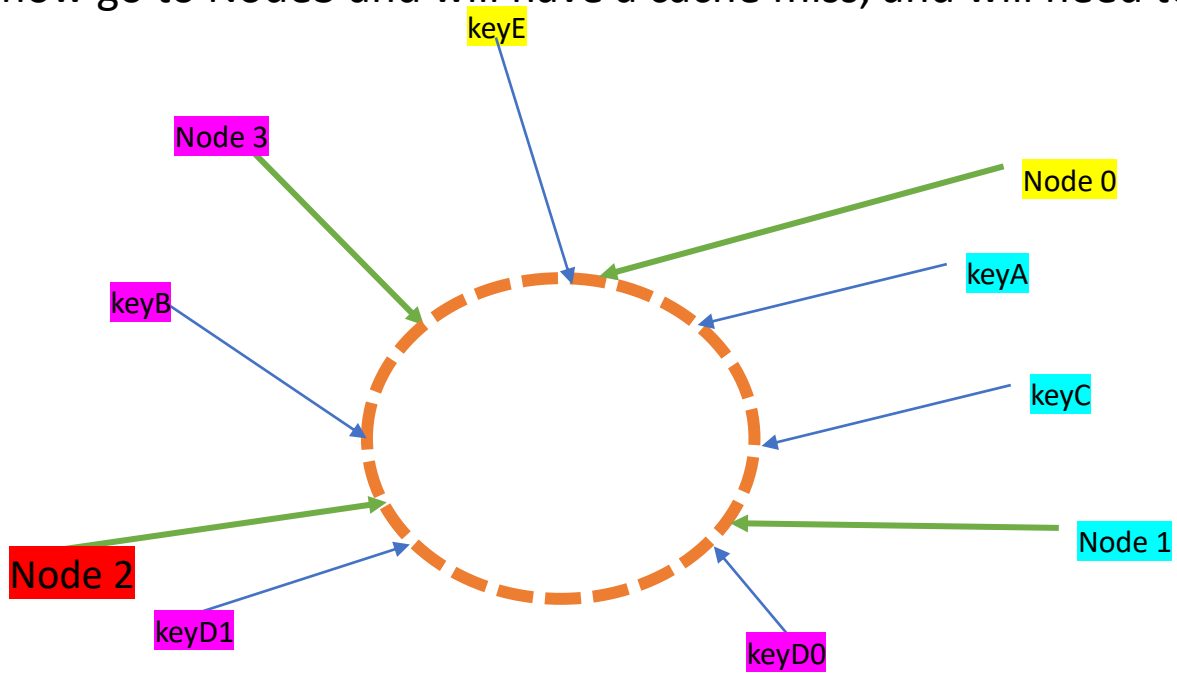
Keys assigned to Node2 would now belong to Node3, since that is the next clockwise adjacent node.

And, the great thing is, this event did not affect any existing keys on Node0, Node1 and Node3.

Note:
We are not necessarily moving the keys from the downed node to Node3 (although that is an option, but that is outside of the “consistent hashing” feature).

If we don’t move the keys, then reads of those keys will now go to Node3 and will have a cache miss, and will need to be populated again (this time in Node 3)

But, do u still see some problem here?



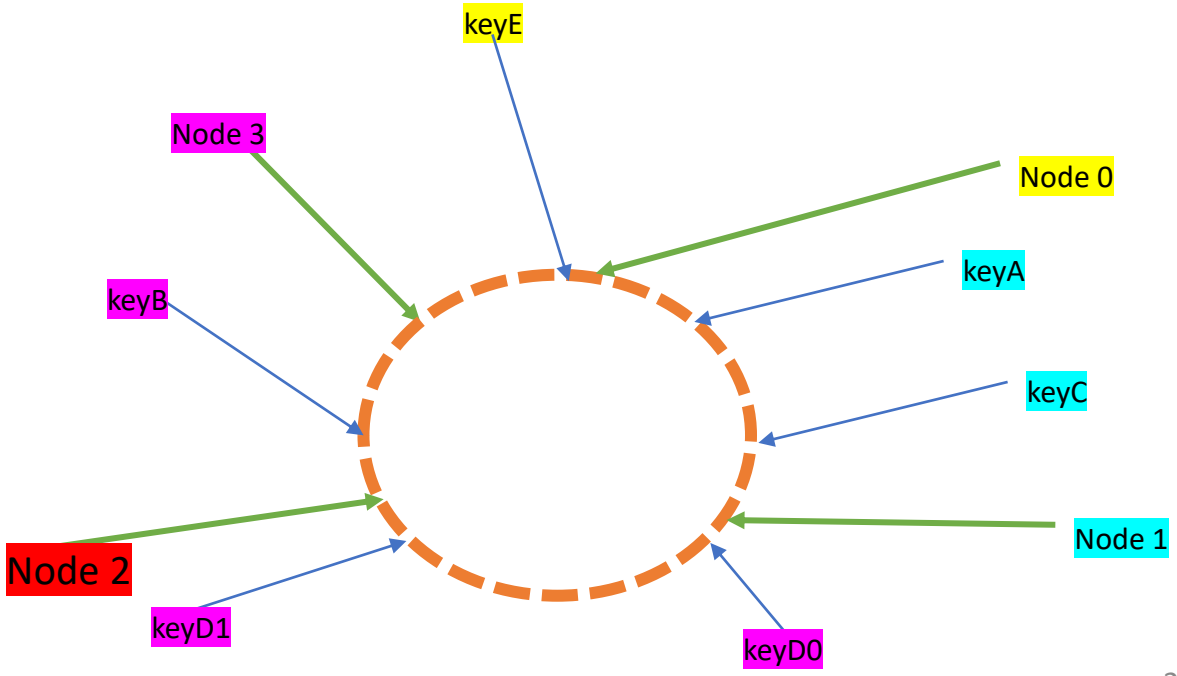
Mapping to servers

The problem we need to deal with is that the load on Node3 would go up a lot, potentially double.

- actually the factor depends on server weight allocation, we will talk about that soon.

When a cache server goes down, if we end up assigning all of its keys to the next node, we will still have a problem because this next node may also go down since we put a lot of data on it.

And this can lead to a cascading effect and bring down the whole distributed system.



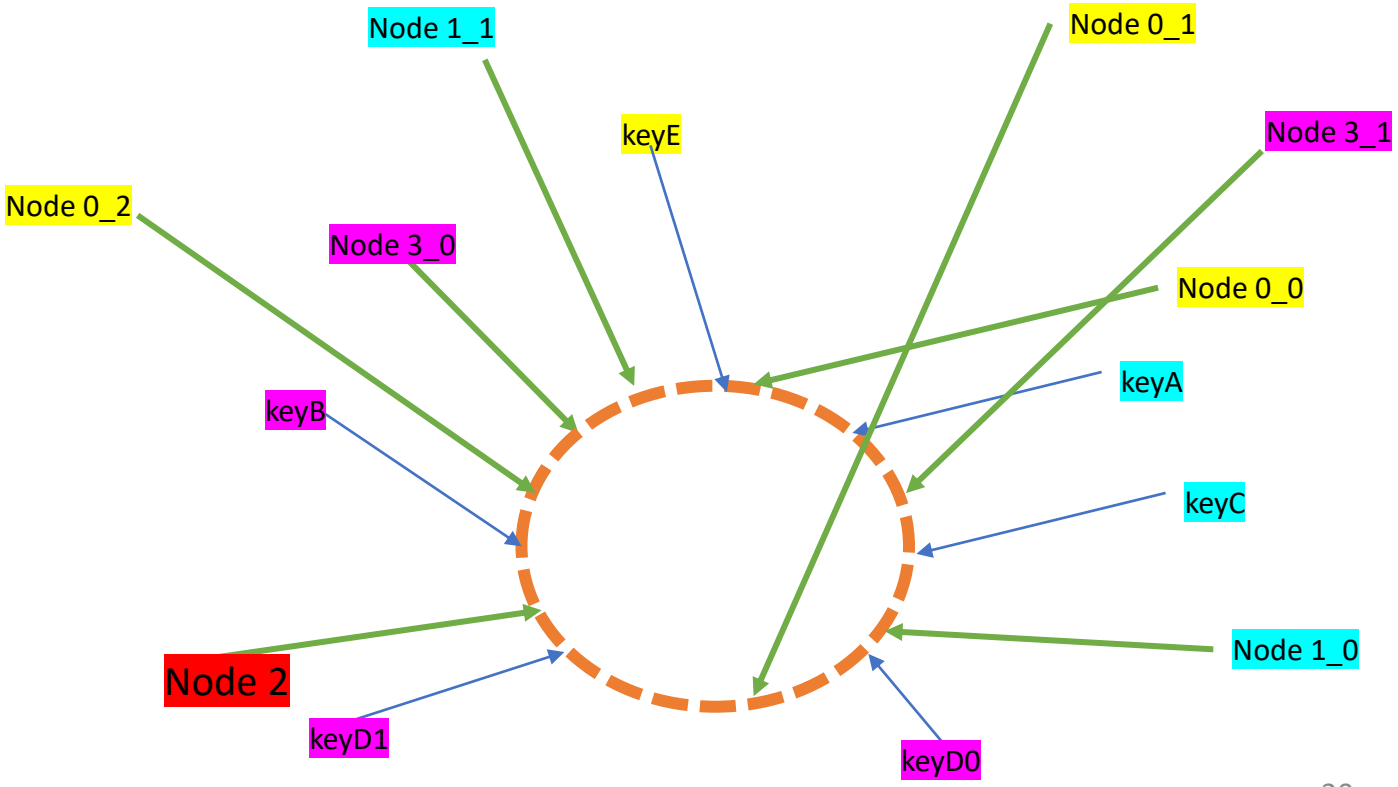
Mapping to servers

So, we add lots of virtual nodes to the hash ring.

🡸 Add more virtual nodes than what I show in the diagram.

This means that if a node goes down, its keys get distributed to a lot more servers than just one.

The keys affected are still only the ones belonging to the server that went down.
But the data distribution does not get skewed.



Mapping to servers

If we have a scenario where servers are of different SKUs, we can adjust the number of virtual nodes added.

For stronger servers … bigger RAM, more cores (if work is CPU bound), we can add more virtual nodes for them, compared to servers not as high end.

That way, the stronger servers will end up getting more data.

complexity

Time

Adding or removing a key: ?

Adding or removing a server ?

Space

complexity

Time

N: number of servers
K: number of keys

Adding or removing a key:	O(1) to add the key, but O (log N) to determine which server	← O (log N)
Adding or removing a server	O(log N) to add/remove the server. Cost of keys affected can be quantified as O (K / N) K / N is the keys per server (assuming fairly equal distribution)	

Space
O(1)