Chapter 5 Design Consistent Hashing

Goals for consistent hashing:

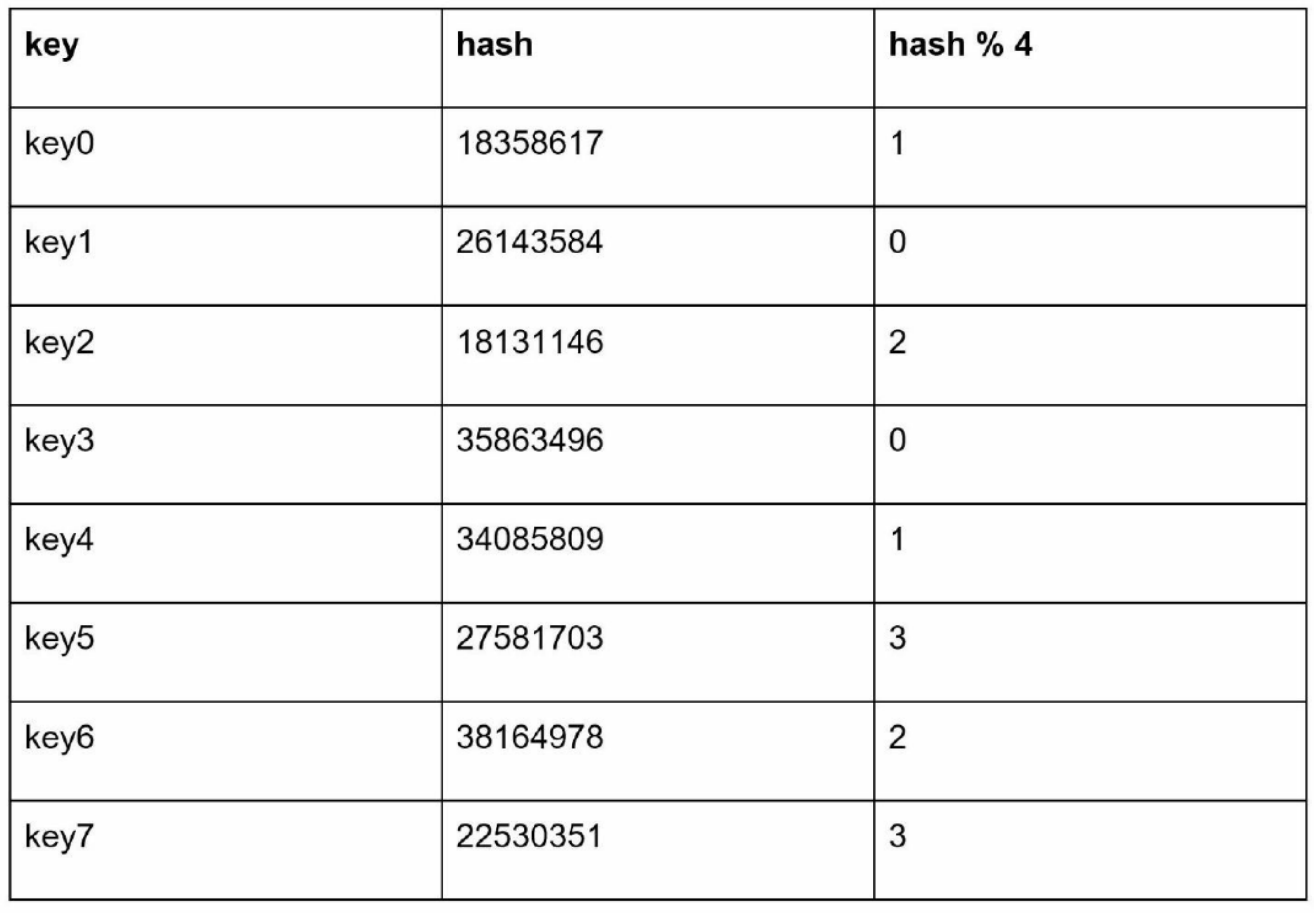
- a commonly used technique for horizontal scaling
- distribute requests/data efficiently and evenly across servers.

The Rehashing Problem

A common way of balance the load for n cache servers:

$$serverIndex = hash(key) \ \% \ N, \text{ where } N \text{ is the size of the server pool}$$

Example:



key	hash	hash % 4
key0	18358617	1
key1	26143584	0
key2	18131146	2
key3	35863496	0
key4	34085809	1
key5	27581703	3
key6	38164978	2
key7	22530351	3

Table 5-1

fetch the server where the key is stored: hash(key) % 4

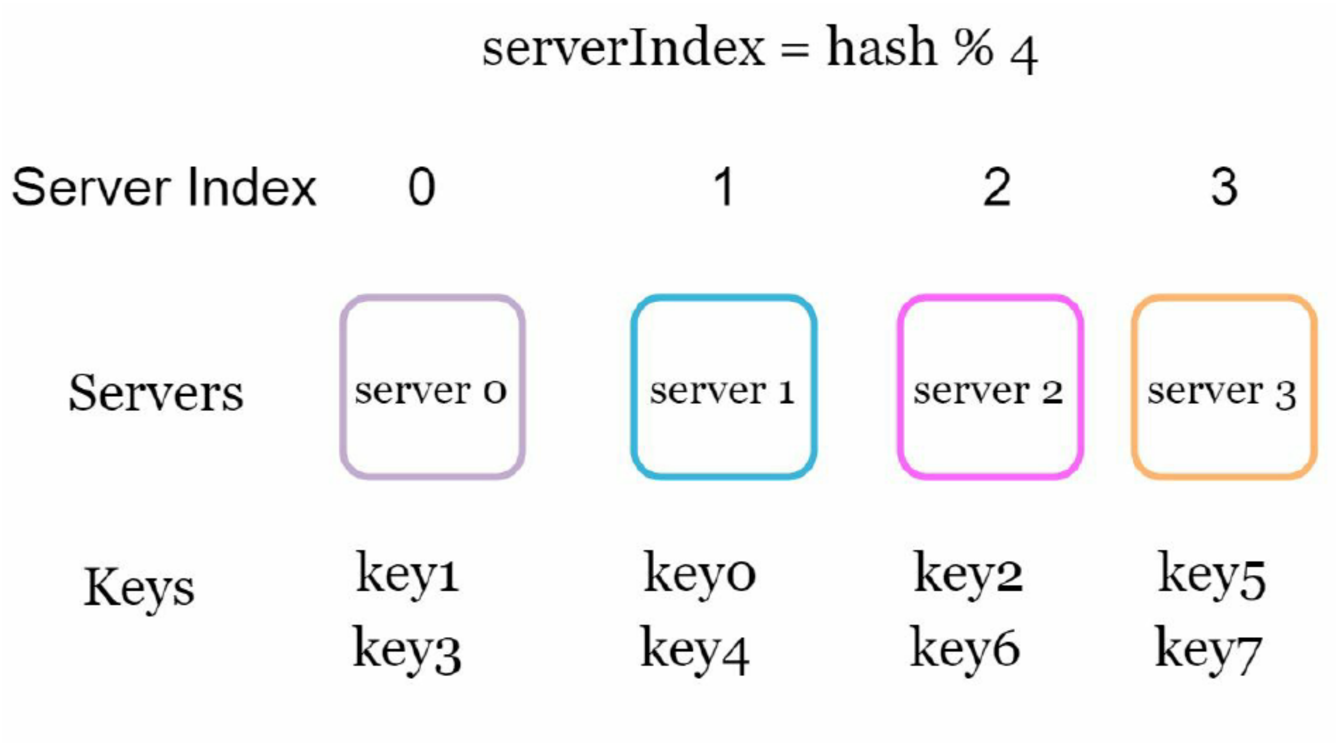


Figure 5-1

Problems of traditional hash tables:

- work well: when size(pool) is fixed, and data distributed evenly
- have problems: when new server is added or any serve is removed
  - server pool size changed
  - hash(key) is the same, but hash(key) % (serve size) changed
  - result: keys are redistributed, not just the ones originally stored in the offline server -> storm of cache misses (most cache client will connect to the wrong servers to fetch data)

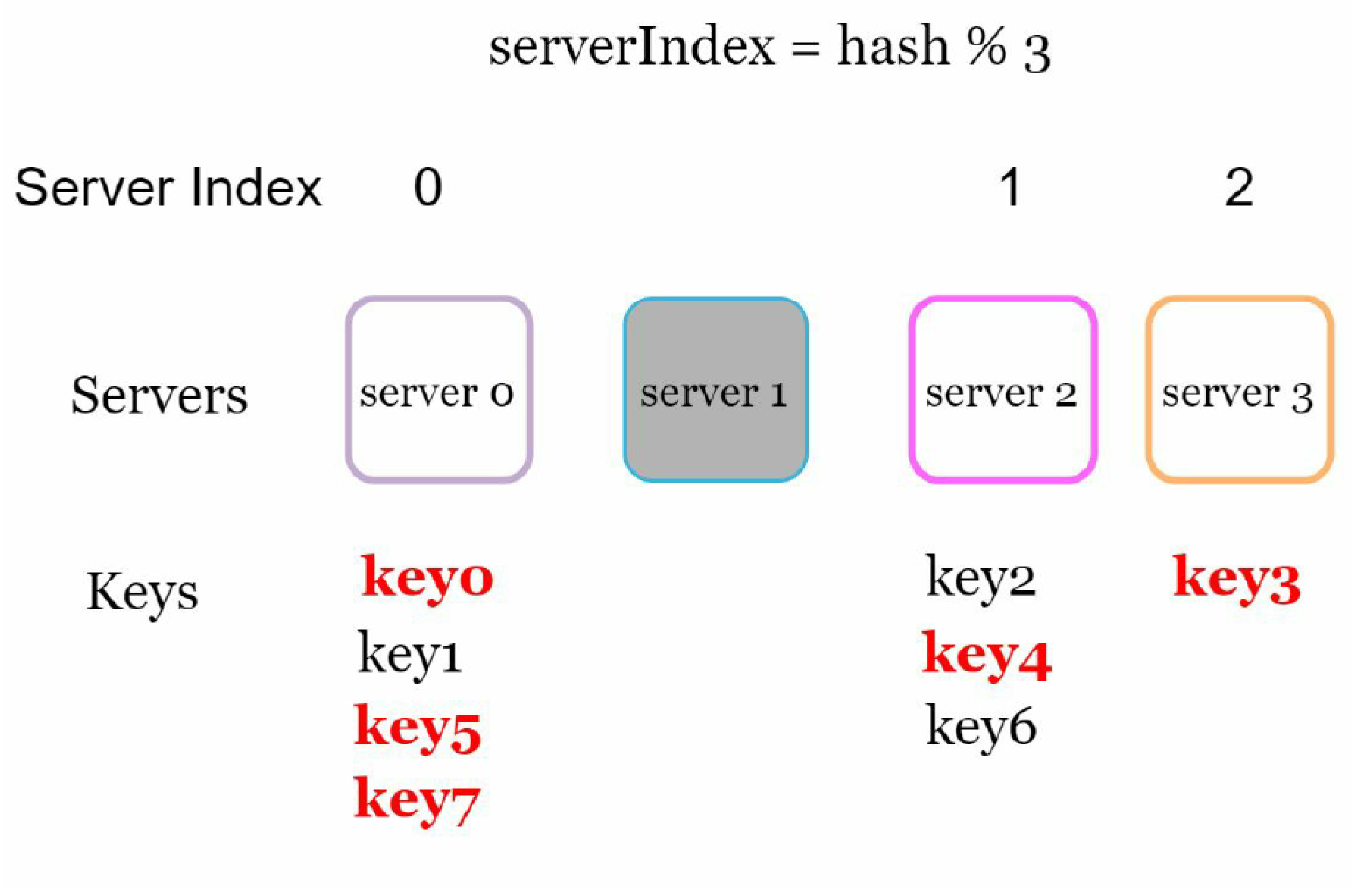


Figure 5-2

Consistent hashing

Wiki definition: Consistent hashing is a special kind of hashing such that when a hash table is re-sized and consistent hashing is used, **only k/n keys need to be remapped on average**, where k is the number of keys, and n is the number of slots. In contrast, in most traditional hash tables, a change in the number of array slots causes nearly all keys to be remapped [1]

Hash space and hash ring

Hash space: the output range of the hash **function**. Assume SHA-1 (Secure Hash Algorithm1, though it is considered insecure now, has been replaced by SHA-3 or SHA-256) is used, the hash space goes from 0 to 2^160 -1

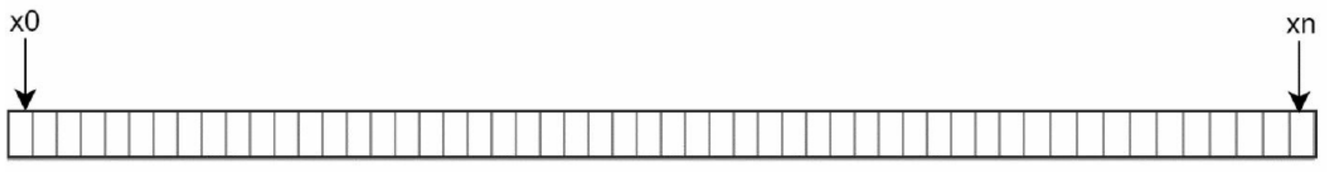


Figure 5-3

Hash ring: collecting both ends of the hash space

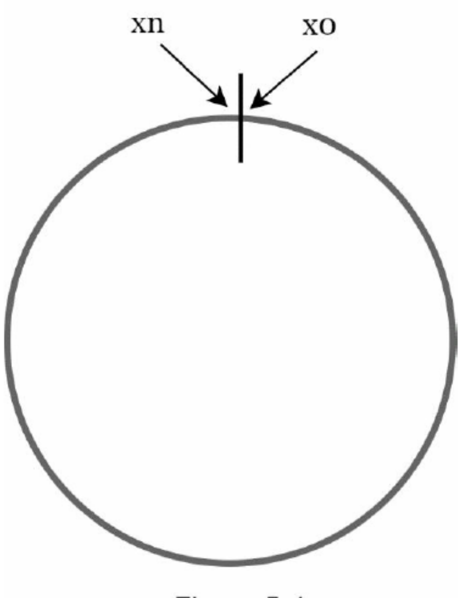


Figure 5-4

Hash servers, Hash keys & Server lookup

Hash servers: Map serves based on server IP or name onto the ring
Hash keys:

- no modular operation: hash function used is different from the one in rehashing problem
- keys are hashed onto the hash ring
  Server lookup: To determine which server a key is stored on, we go **clockwise** from the key

position on the ring until a server is found.

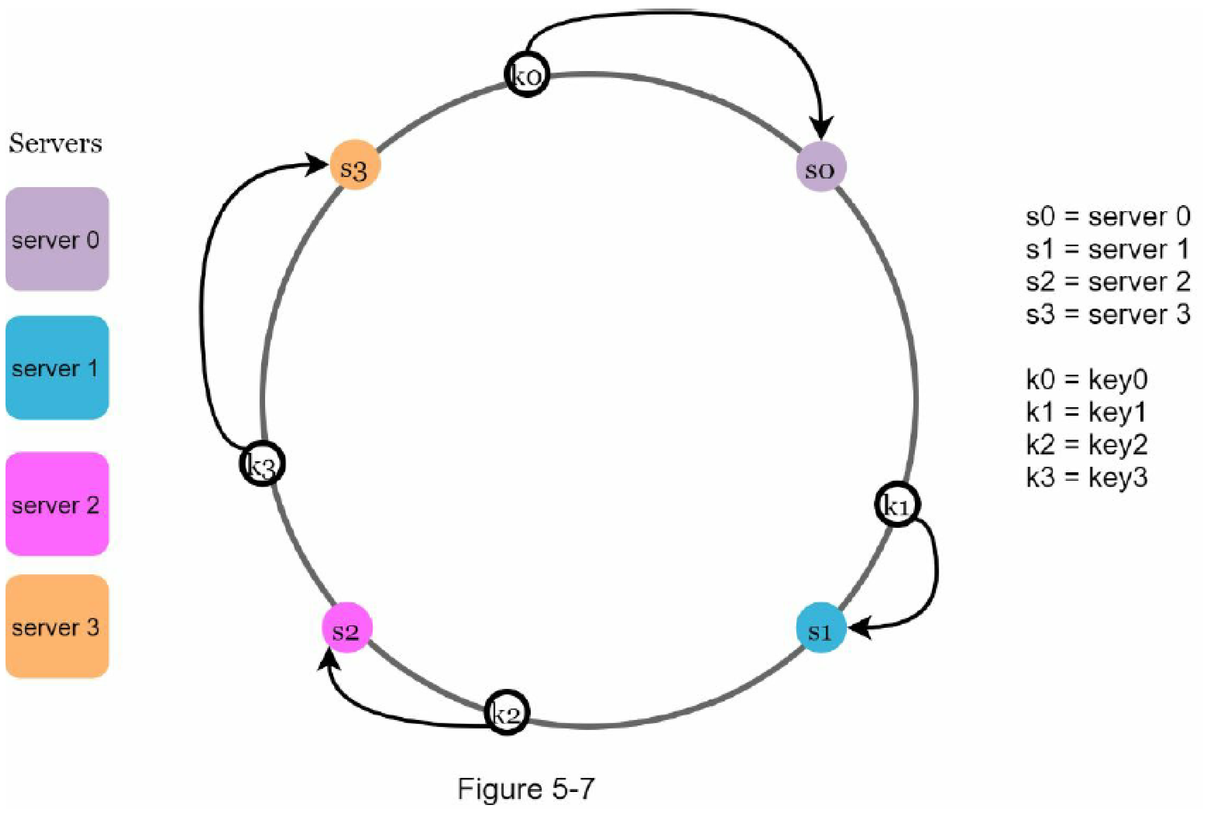


Figure 5-7

Add or Remove a server

Add a server: Adding a new server will only require distribution of a fraction of keys. E.g.: key0 from old s0 to newly added s4

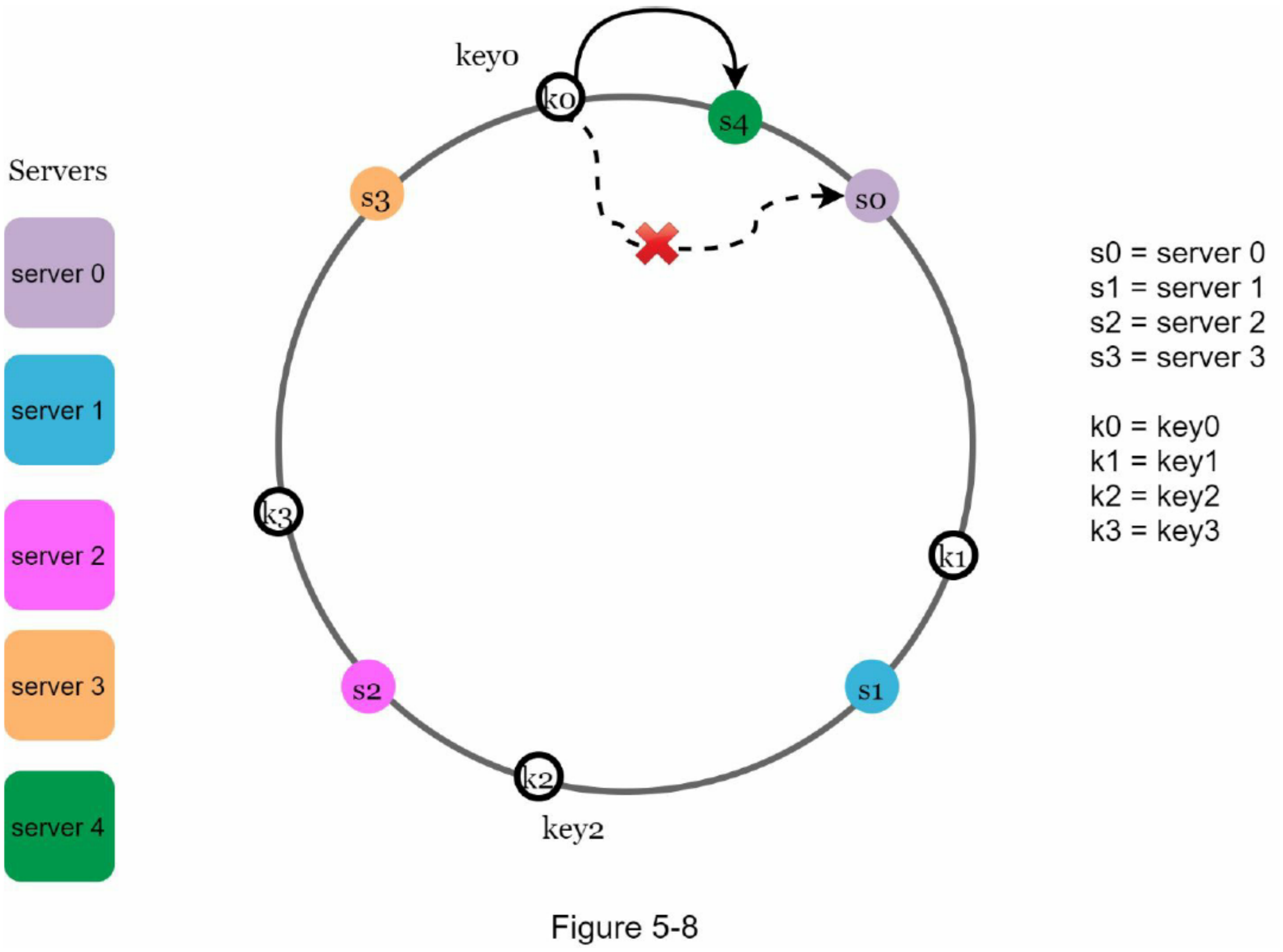


Figure 5-8

Remove a server: also a small fraction of keys requires redistribution

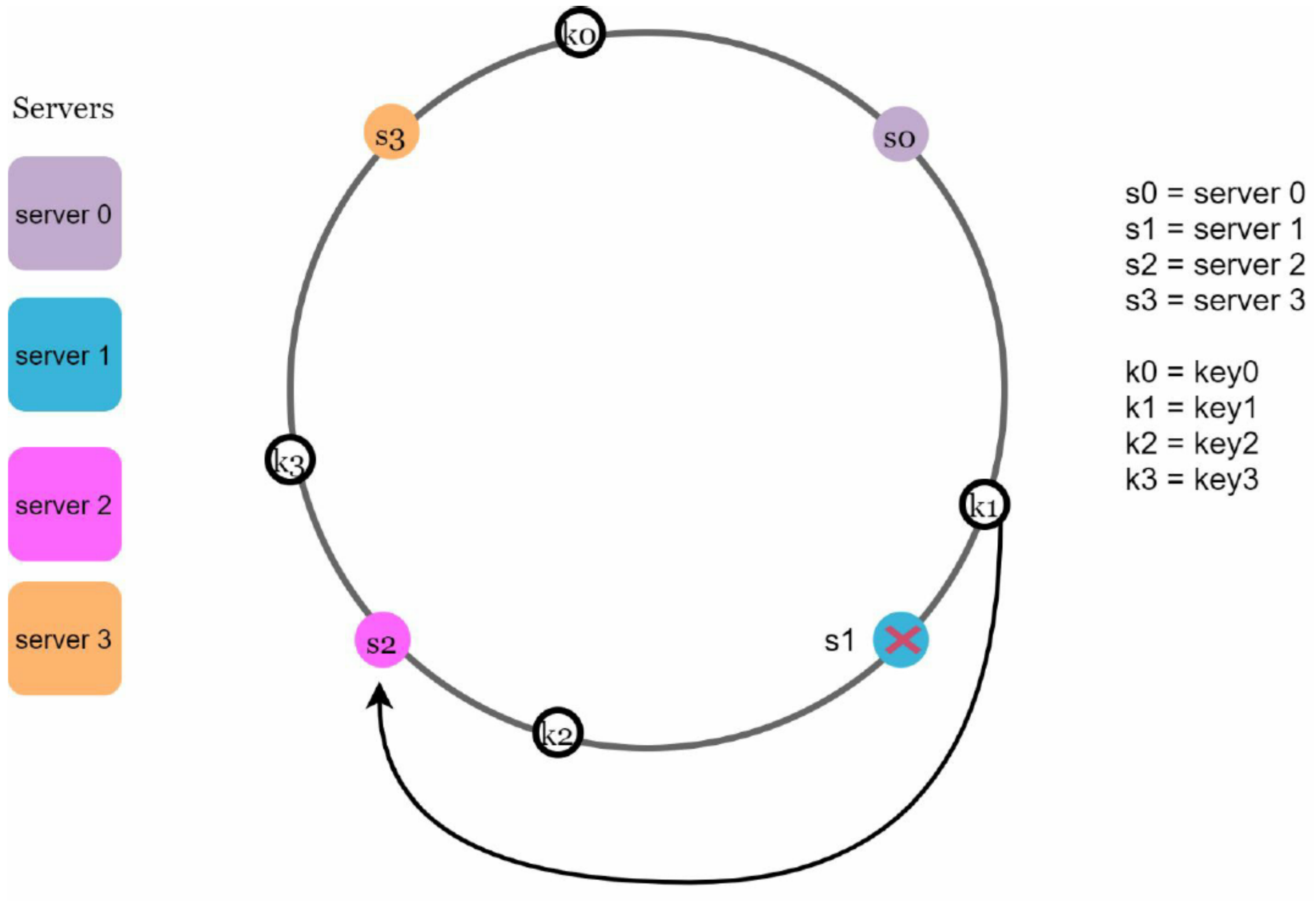


Figure 5-9

Issues of the basic approach

Basic steps of **Consistent Hashing Algorithm** (by Karger et al.):

- Map servers and keys on to the ring using a uniformly distributed hash function.
- To find out which server a key is mapped to, go clockwise from the key position until the first server on the ring is found

Two problems:

- It is **impossible to keep the same size of partitions** (the hash space between adjacent servers) on the ring for all servers considering a server can be added or removed.

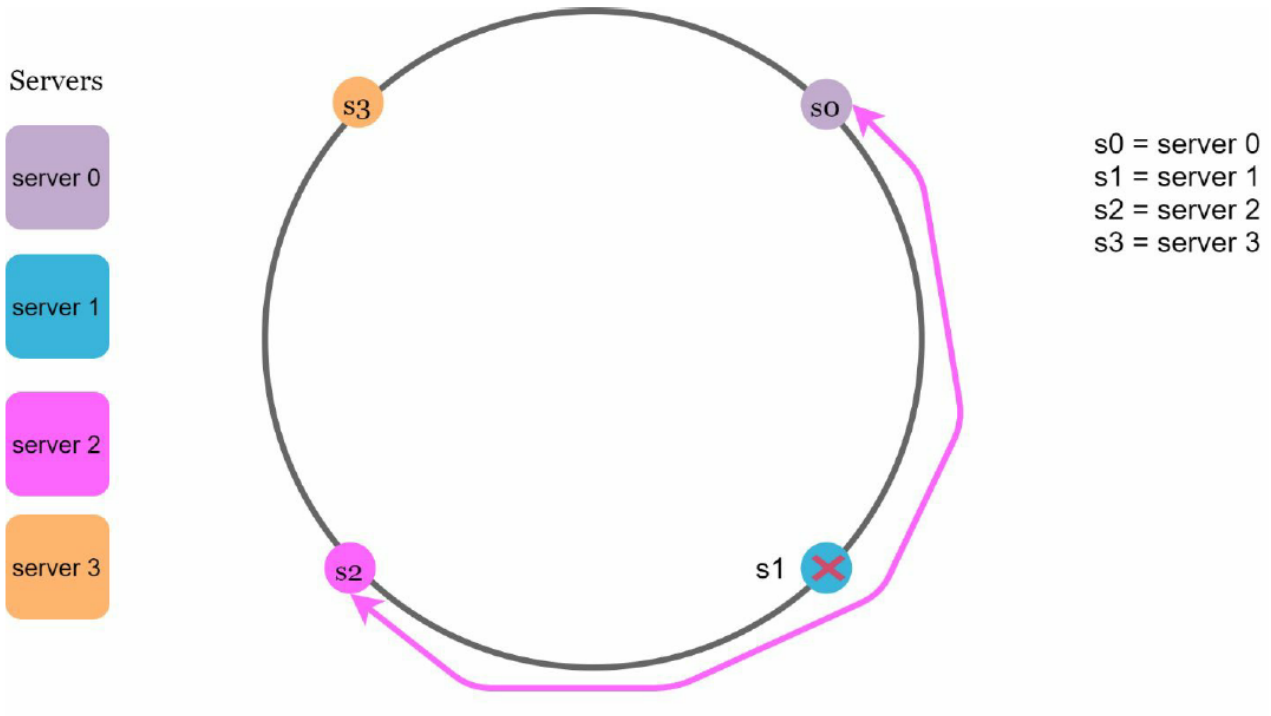


Figure 5-10

- It is possible to have a **non-uniform key distribution** on the ring.

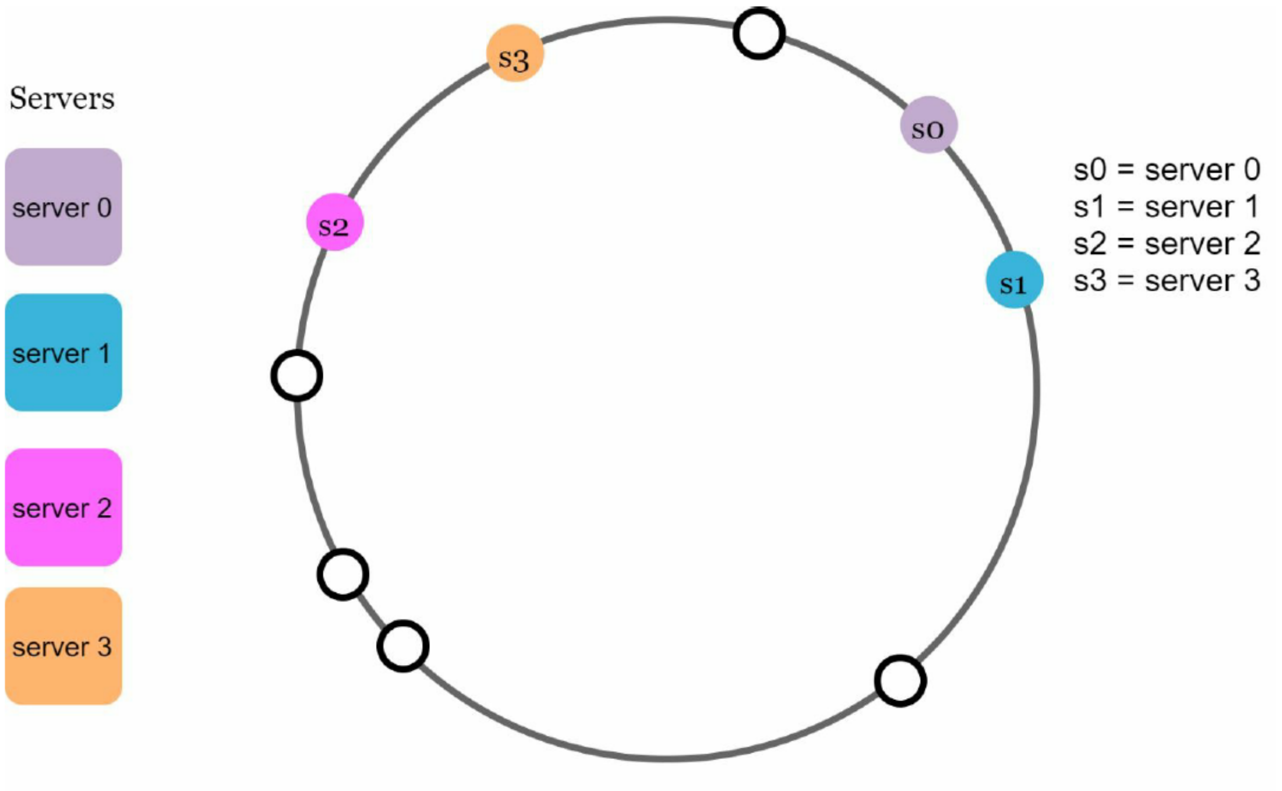


Figure 5-11

Solution: virtual nodes or replicas

Virtual nodes

virtual node: A virtual node refers to the real node, and each server is represented by multiple virtual nodes on the ring.

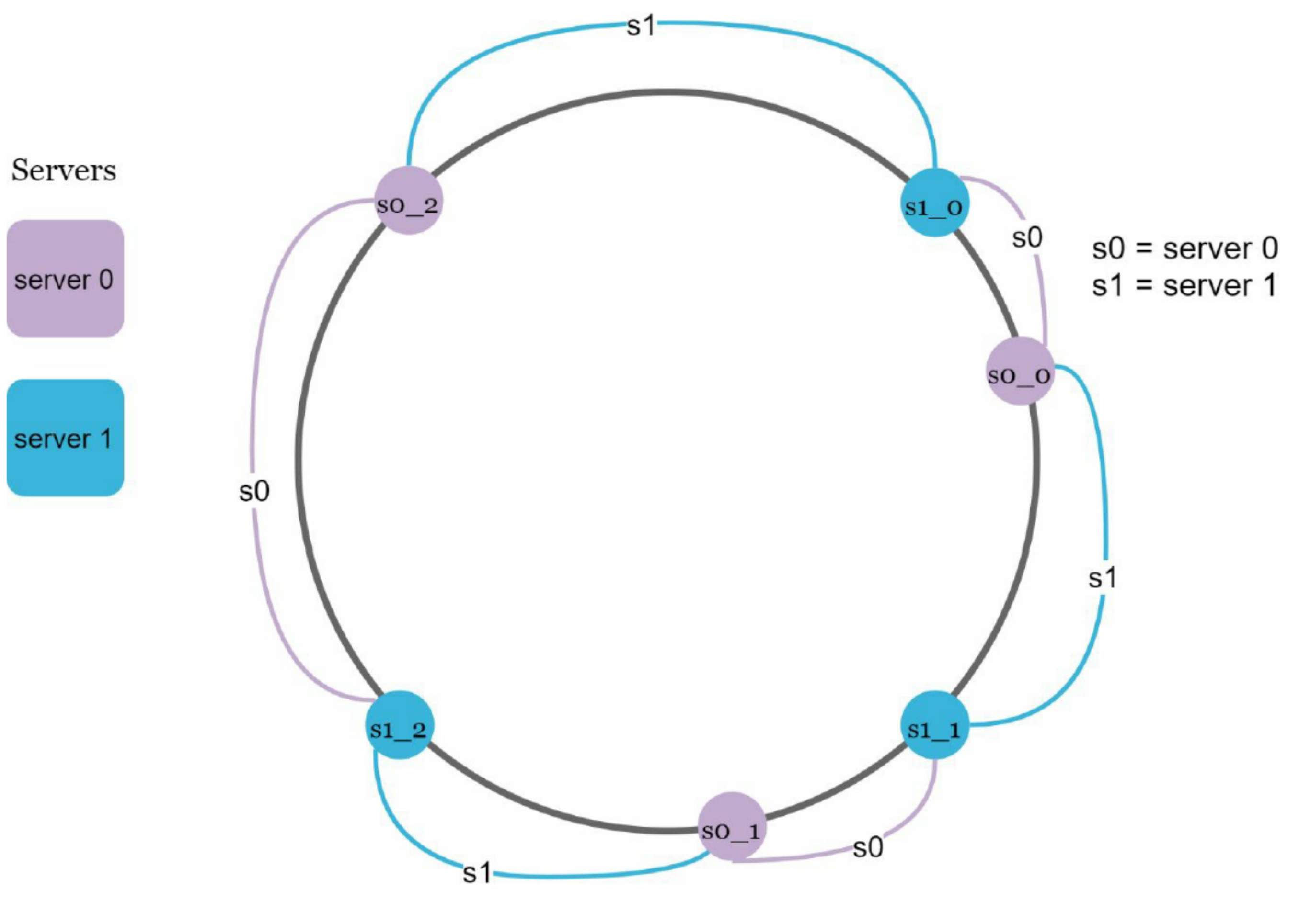


Figure 5-12

- In **real world**, the **number of virtual nodes is much larger** than the example above
- **Each server** is responsible for **multiple partitions**
- To find which server a key is stored on, we go clockwise from the key's location and **find the first virtual node** encountered on the ring, and then map to real server
- As the number of virtual nodes increases, the **distribution of keys becomes more balance**d, because **standard deviation gets smaller** with more virtual nodes.
- **Trade-off** of increasing the number of virtual nodes:
  - standard deviation will be smaller
  - space that needed to store data about VD will be more
  - Tune the number of ND to fit system requirements

Find affected keys

Find the affected range to redistribute the keys:

- **When a new server is added**: The affected range starts from the newly added node and moves anticlockwise around the ring until a server is found

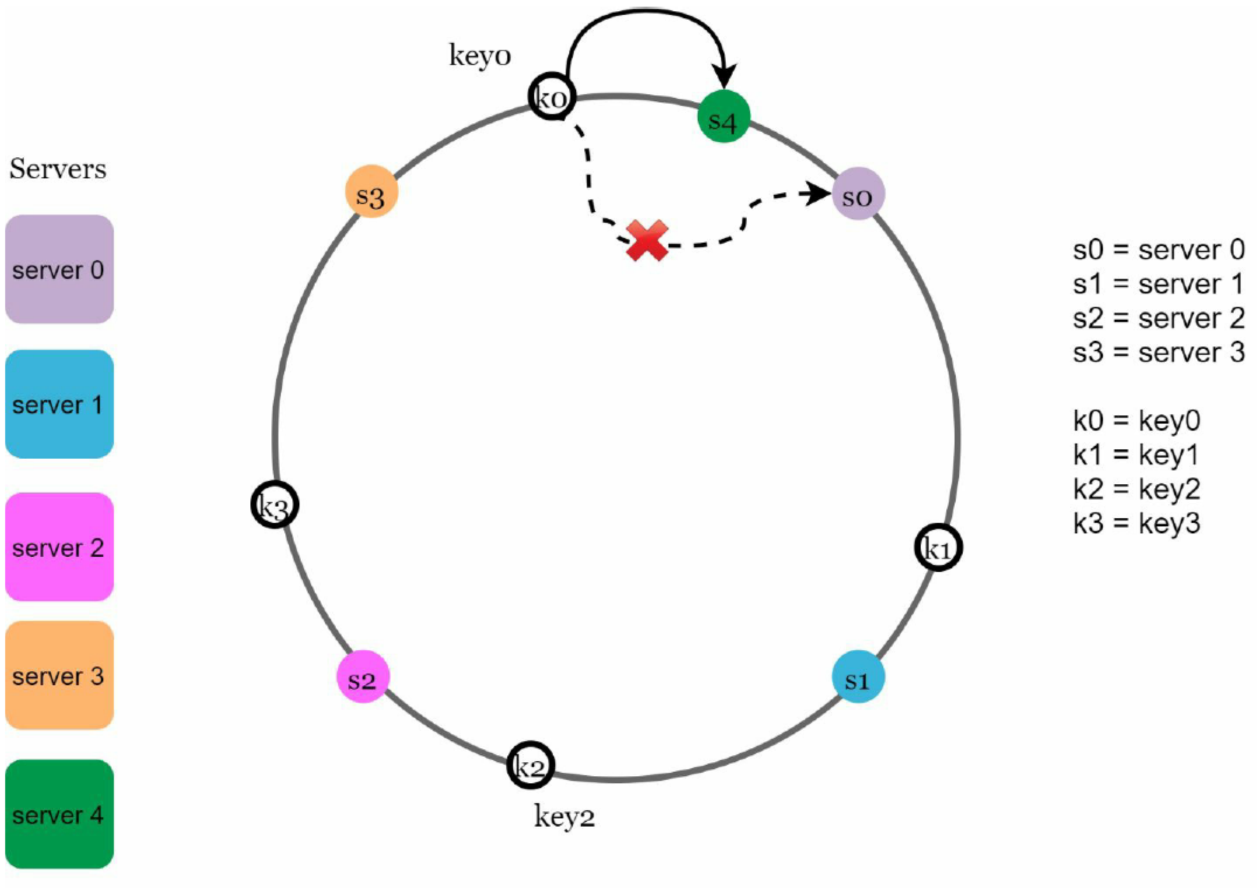


Figure 5-14

- **When a server is removed**: the affected range starts from the removed node and moves anticlockwise around the ring until a server is found.

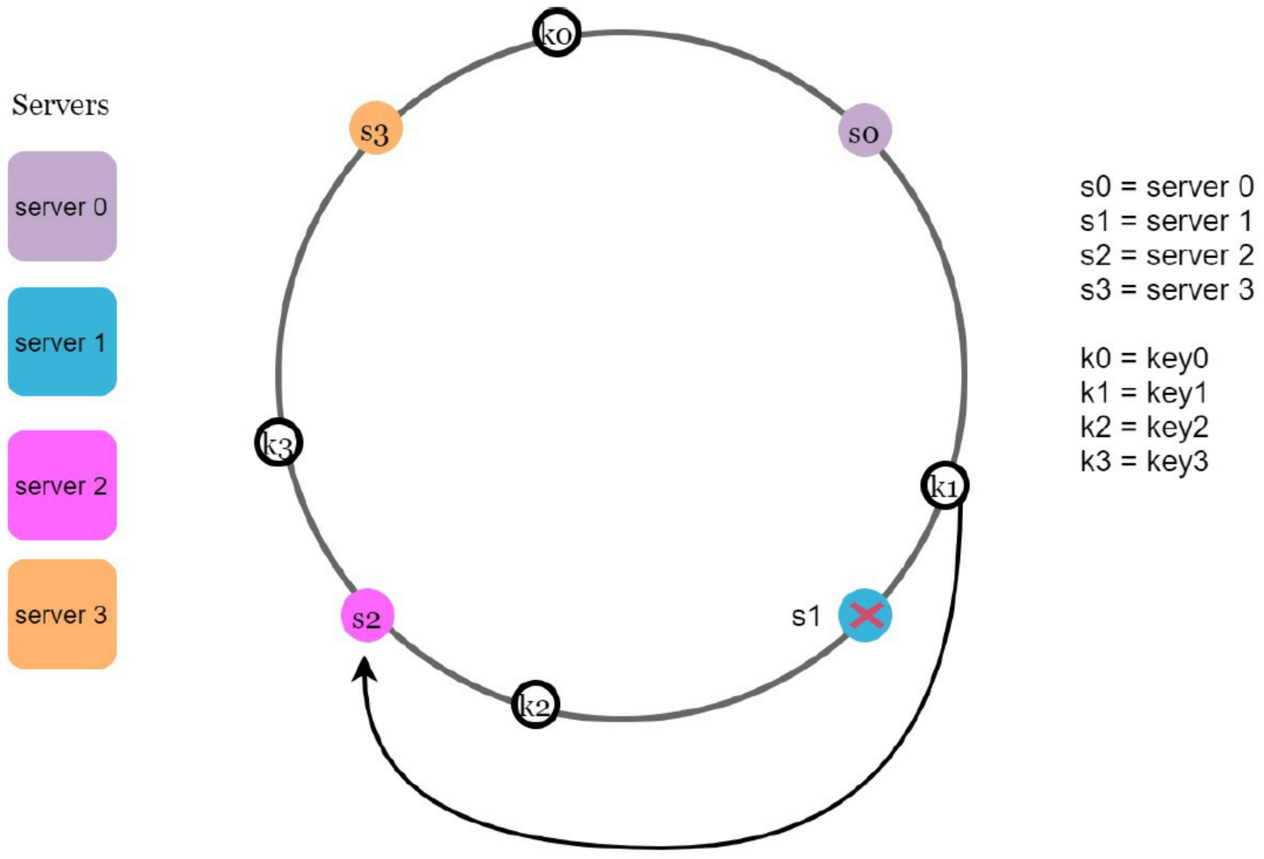


Figure 5-15

Wrap-up

The **benefits** of consistent hashing:

- **Fault Tolerance and reduced downtime**: **Minimized keys are redistributed** when servers are added or removed.
- **Scalability**: It is **easy to scale horizontally** because data are more evenly distributed.
- **Decentralization**: It allows a decentralized control mechanism, where each node can independently determine where a key should be stored or retrieved from without relying on a central coordinator
- **Mitigate hotspot key problem**. Excessive access to a specific shard could cause server overload. Imagine data for Katy Perry, Justin Bieber, and Lady Gaga all end up on the same shard. Consistent hashing helps to mitigate the problem by distributing the data more evenly.

Some notable real-world systems:

- Partitioning component of Amazon's Dynamo database
- Data partitioning across the cluster in Apache Cassandra
- Discord chat application

- Akamai content delivery network
- Maglev network load balancer

Limitations of Consistent Hashing:

- **Hot Spots**: Even with virtual nodes, consistent hashing can still lead to hot spots where certain nodes handle a disproportionat ely high load, e**specially if the key distribution is not uniform**
- **Complexity** in Implementation: Implementing consistent hashing correctly can be complex, particularly when **dealing with virtual nodes, replication, and ensuring proper load balancing**.
- **Node Heterogeneity**: **Inconsistent performance across nodes** (e.g., due to different hardware capabilities) can lead to suboptimal load distribution and can require additional mechanisms to account for node heterogeneity.
- **Maintenance Overhead**: Managing the ring structure, virtual nodes, and handling node failures or additions can **introduce maintenance overhead and require careful tuning.**
- **State Consistency**: **Ensuring consistent state across nodes during dynamic changes** (node addition/removal) can be challenging and may require sophisticated algorithms to **handle state synchronization and data integrity**.
- **Limited to Hashable Keys**: Consistent hashing relies on the ability to hash keys, which **may not be applicable for all types of data or applications**, limiting its use cases to those where keys can be effectively hashed.

Consistent Hashing SD Interview Questions

Easy Questions

- Explain what consistent hashing is and how it works
- How would you implement consistent hashing for a basic caching system?
    - Expected Answer: Outline the steps to implement consistent hashing, including creating a hash ring, mapping keys to nodes, and handling node additions/removals

Medium Questions

- **Virtual Nodes**: What are virtual nodes in consistent hashing, and why are they used? How would you implement them?
    - Expected Answer: Explain the concept of virtual nodes, their purpose in balancing load, and the implementation details, including how many virtual nodes to create and how to map them to physical nodes.
- **Handling Node Failures**: How does consistent hashing handle node failures? Describe the process and any potential challenges.

    - Expected Answer: Discuss the re-mapping of keys when a node fails, the minimal movement of keys to other nodes, and strategies to ensure data redundancy and consistency.
- **Load Balancing:** How can consistent hashing be optimized to ensure even load distribution across nodes?
    - Expected Answer: Talk about using virtual nodes, monitoring node loads, dynamically adjusting the number of virtual nodes, and potential load balancing algorithms.

Hard Questions

- **Design a Distributed Database**: Design a distributed database system using consistent hashing. Consider aspects such as data partitioning, replication, fault tolerance, and scalability.
    - Expected Answer: Provide a comprehensive design, including the use of consistent hashing for data partitioning, replication strategies (e.g., primary-replica, quorum-based), fault tolerance mechanisms (e.g., data recovery, redundancy), and how to handle node joins/leaves while maintaining data consistency.
- **Dynamic Scaling**: How would you design a system that dynamically scales the number of nodes based on load using consistent hashing? Discuss the challenges and your approach
    - Expected Answer: Describe a system that monitors node load, automatically adds/removes nodes based on predefined thresholds, handles the rebalancing of keys with minimal disruption, and ensures data integrity and consistency during scaling operations.
- **Multi-Datacenter Consistency**: Design a multi-datacenter system that uses consistent hashing to ensure high availability and fault tolerance. How would you handle cross-datacenter consistency and latency?
    - Expected Answer: Discuss a design that uses consistent hashing within each datacenter, replicates data across datacenters for fault tolerance, and employs techniques such as eventual consistency, conflict resolution, and optimized data synchronization to handle latency and ensure data consistency across locations.