

Advanced Distributed Systems

Higher Institute for Applied Sciences and Technology

Student: ضياء حنا
Teacher: محمد بشار دسوقي
Date: 3/4/2026

Consistent Hashing Report

April 3 2026

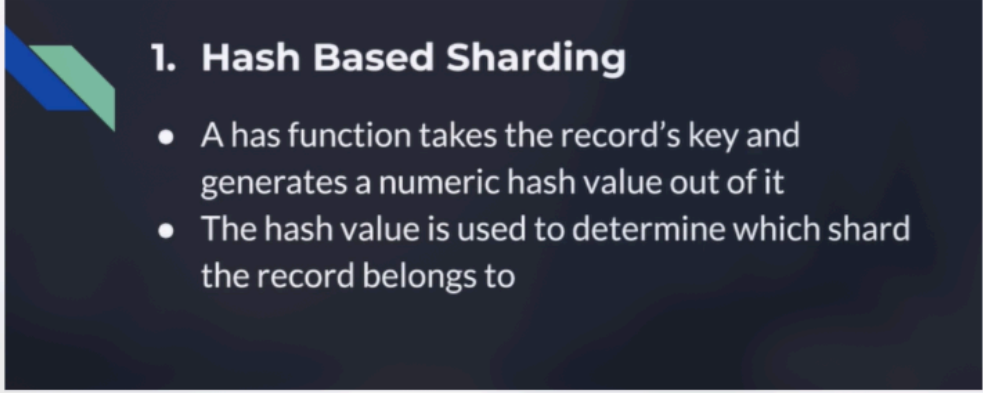
This file is divided into tasks each task will talk about specific thing about this homework

Task 1 (Understanding what Consistent Hashing is All about)

As we saw in lab class

We have multiple strategies to distribute data object (key-value) pairs among shard the first one we learned is hash based sharding

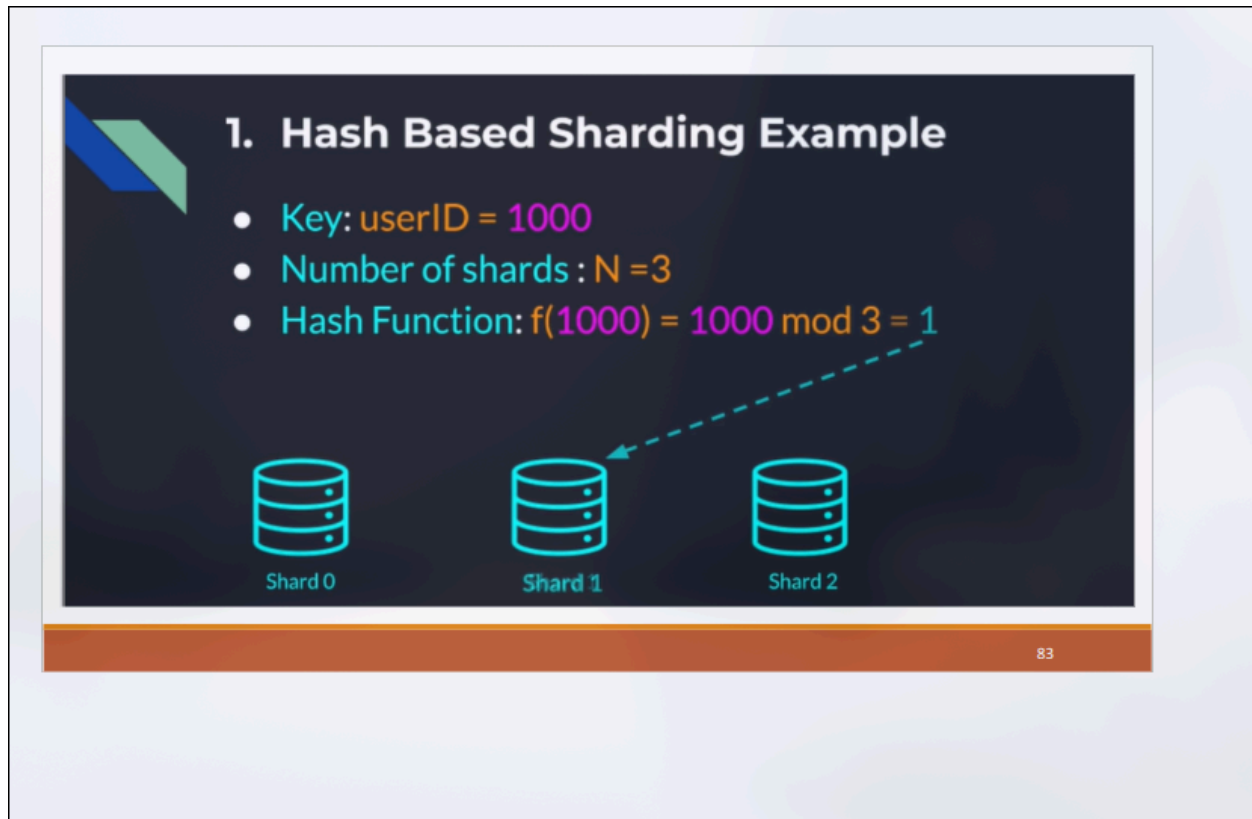
Which is defined in this slide



1. Hash Based Sharding

- A has function takes the record's key and generates a numeric hash value out of it
- The hash value is used to determine which shard the record belongs to

82



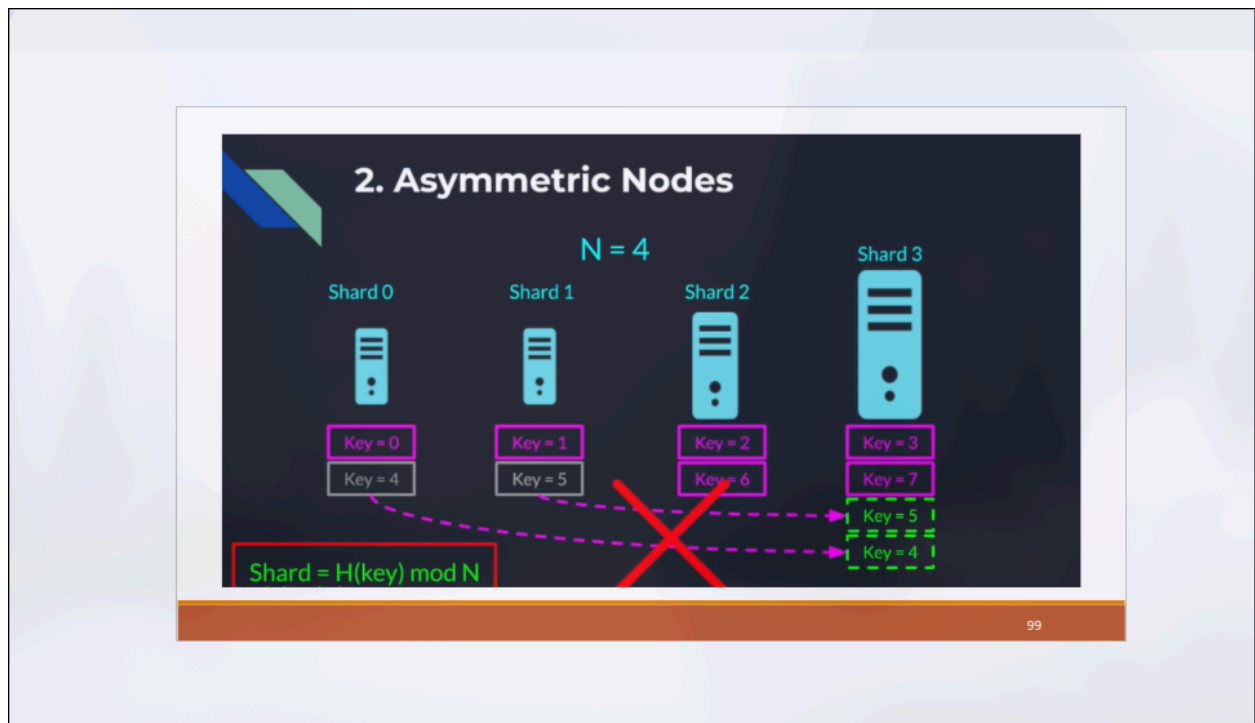
These methods have the advantage of being even (each shard gets approximately the same number of object).

It also has no weaknesses the first one is



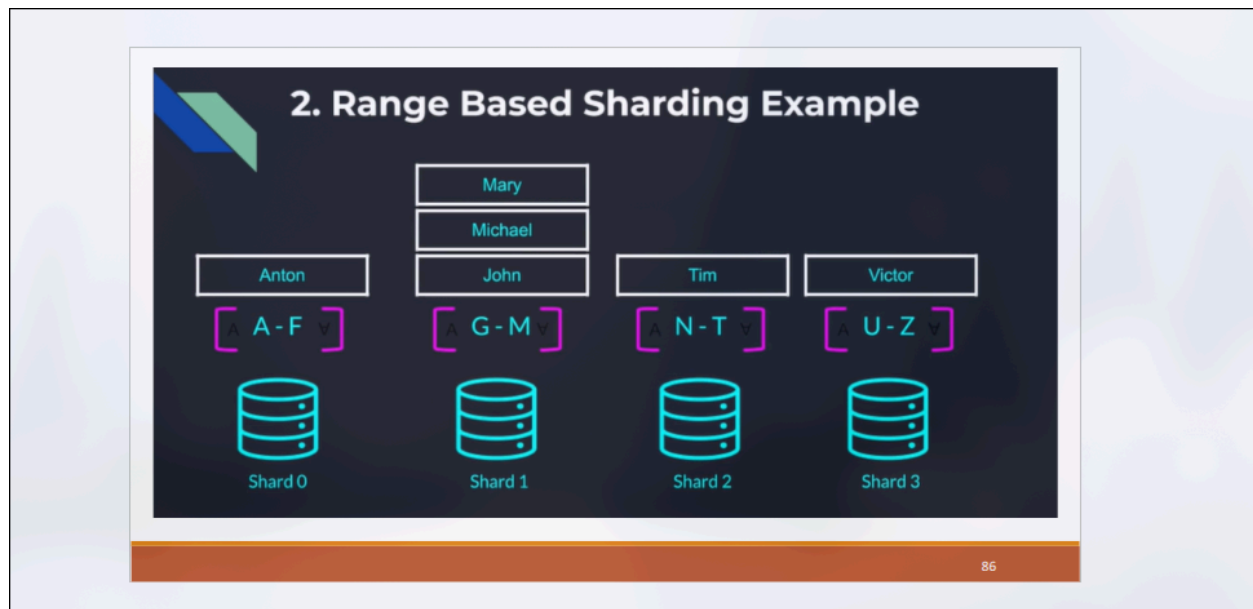
Scaling and adding or deleting new nodes don't work well because key rebalancing among the remaining nodes require too much computation for all the keys

The second problem is asymmetric nodes



Which means sometimes some nodes have more computational power than others which means we don't need even distribution in this case.

We also learned about range based sharding



This algorithm is simple and fast but it has a weakness of being unfair (uneven) .One key subspace might hold a lot of objects compared to others.

Because the previous two algorithms both have weaknesses people had to invent more complex algorithms that solve these weaknesses somehow

The new algorithm that we took in class is

called consistent hashing

Found this really good website that explains it in a very good way

[“https://highscalability.com/consistent-hashing-algorithm/”](https://highscalability.com/consistent-hashing-algorithm/)

Asking chatGPT to summarize what is consistent hashing algorithm we get the following answer

“Consistent hashing is a way to **distribute data across multiple servers/nodes** so that when a server is **added or removed**, **only a small portion of the data moves**.

That’s the whole reason it exists.

Why normal hashing is annoying

Suppose you store cache/data on 4 servers:

$\text{server} = \text{hash}(\text{key}) \% 4$

Example:

$\text{hash}(\text{"user123"}) \% 4 = 2$

So **"user123"** goes to **server 2**.

Problem:



If you add one more server:

$\text{server} = \text{hash}(\text{key}) \% 5$

Now **almost everything changes**.

That means:

- cached sessions move
- cache misses explode
- load shifts badly
- distributed systems become painful

So normal modulo hashing is **not stable** when the number of servers changes.

Consistent hashing idea

Instead of hashing keys directly with $\% N$, we imagine a **big circle** (called a **hash ring**).

Step 1: Hash servers onto the ring

10

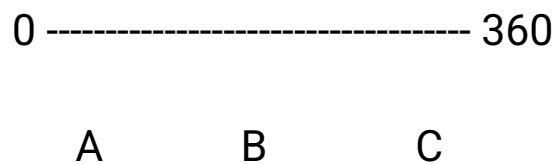
Each server gets a position:

$\text{hash}(\text{ServerA}) \rightarrow \text{position on ring}$

$\text{hash}(\text{ServerB}) \rightarrow \text{position on ring}$

$\text{hash}(\text{ServerC}) \rightarrow \text{position on ring}$

Example ring:



(Real systems use a huge hash space, like $0 \dots 2^{32}-1$ or $0 \dots 2^{128}-1$.)

Step 2: Hash each key onto the same ring

Example:

hash("user123") -> some position

Now place that key on the ring.

Step 3: Walk clockwise until you hit a server

The key belongs to the **first server clockwise** from its hash position.

Example

Ring:

0 ----- 360

A(50) B(180) C(300)

Keys:

- "apple" → 70
- "banana" → 250
- "cat" → 320

Assignment:

- apple (70) → next clockwise server = **B**
- banana (250) → next clockwise server = **C**
- cat (320) → wraps around → **A**

So:

apple -> B

banana -> C

cat -> A

Why this is good

13

Now suppose we add a new server:

D(220)

New ring:

0 ----- 360

A(50) B(180) D(220) C(300)

What changes?

Only keys in the region:

(180, 220]

move from **C** to **D**.

Important:

Most keys stay where they are.

That's the magic.

Main benefit

When the system changes:

- **Adding a node** → only nearby keys move
- **Removing a node** → only that node's keys move
- **Minimal reshuffling**

This is why it's used in:

- distributed caches
 - distributed databases
 - sharding
 - CDNs
 - load balancers
 - peer-to-peer systems
-

Intuition

Think of it like a **pizza** cut into slices.

Each server owns the slice of the ring **between the previous server and itself**.

A key lands somewhere on the pizza and belongs to the next server clockwise.

When you add a new server, you only steal **one slice**, not the whole pizza.

Much less drama.

Problem: uneven distribution

Basic consistent hashing can be **unbalanced**.

Example:

A at 10

B at 20

C at 300

Then **C owns a giant chunk** of the ring.

So one server might get way more keys than others.

Solution: Virtual Nodes (very important)

Instead of putting each real server on the ring **once**, put it **many times**.

Example:

A1, A2, A3

B1, B2, B3

C1, C2, C3

Each one is hashed separately.



17

So the ring becomes more mixed:

A1 C1 B1 A2 C2 B2 A3 B3 C3

Now keys spread much more evenly.

Benefits of virtual nodes:

- better load balancing
- smoother scaling
- easier handling of different server capacities

If one machine is stronger, you can give it **more virtual nodes**.


Replication with consistent hashing

Many systems don't store data on just one node.

Instead:

- primary node = first clockwise server
- replicas = next 2 or 3 servers clockwise

So a key might be stored on:



Key X -> A, then B, then C

This helps with:

- fault tolerance
- availability
- recovery

This is common in systems like:

- Apache Cassandra
 - Riak
 - Amazon Dynamo (concept/design)
 - some distributed cache systems
-

Real-world use cases

1) Distributed cache

Suppose you have 10 cache servers.

For a cache key like:

session:9381

You use consistent hashing to decide which cache node stores it.

If you add an 11th server:

- you don't lose the entire cache
- only some keys remap

This is why it's useful in systems like:

- Memcached client-side sharding
- some Redis cluster designs (though Redis Cluster uses slot-based partitioning, which is related but not identical)

2) Distributed databases

A database cluster needs to decide where a row/document lives.

Consistent hashing helps place data across nodes while keeping movement small when scaling.

3) Load balancing

Some systems hash:

- user ID
- session ID
- IP
- request key

so the same user/request tends to hit the same backend.

Time complexity

If servers are kept in a **sorted structure** (like a balanced tree / sorted array):

- find server for key = $O(\log N)$
(binary search on ring positions)

With good implementation, it's very efficient.

Tiny example in plain logic

Imagine this ring:

Servers:

A = 100

B = 400

C = 700

Key:

`hash("dog") = 450`

Walk clockwise:

- $450 > 400$
- next server is $700 = \mathbf{C}$

So:

`dog -> C`

If key was:

$\text{hash}(\text{"zebra"}) = 900$

There is no server after 900, so wrap around to the first one:

zebra -> A

Difference from “simple hashing”

Simple hashing:

$\text{hash}(\text{key}) \% \text{number_of_servers}$

Bad:

- adding/removing servers remaps almost everything

Consistent hashing:

place keys and servers on same ring

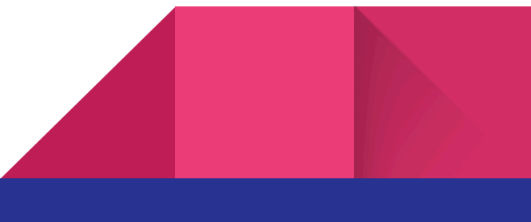
pick next clockwise server

Good:

- minimal remapping
- scalable
- ideal for dynamic clusters

One-line definition

Consistent hashing is a partitioning technique that maps both nodes and keys into the same hash space so that cluster changes cause only minimal key redistribution.



Super short exam-style answer

Consistent hashing is a data distribution algorithm used in distributed systems to assign keys to servers in a way that minimizes remapping when servers are added or removed. It works by hashing both servers and keys onto a circular hash space (hash ring), and assigning each key to the next server clockwise on the ring.

If you want, I can also give you:

1. a **visual diagram**,
2. a **step-by-step worked example**, or
3. **how it's implemented in code (C / Python / Java)**.

”



Task 2(Understanding The Code)

The code is well documented and with a little help of chatGPT To understand the code

This task can be done easily

I shared the code explanation chatGPT session in this file

["https://chatgpt.com/share/69d49370-10ec-8326-914d-3f35485cf6a0"](https://chatgpt.com/share/69d49370-10ec-8326-914d-3f35485cf6a0)

This session may also be useful for other students to read if they are struggling with understanding the code

Task 3(running the code)

Following the instruction of build_and_run.md

We can execute it in the terminal

```

cache-01: 995 items (24.9%)
cache-02: 1022 items (25.6%)
cache-03: 1013 items (25.3%)
cache-04: 970 items (24.3%)

✓ System automatically redistributed load to recovered node

DEMO 6: LOAD BALANCER
Route user sessions to web servers based on user ID

Web servers: web-server-1, web-server-2, web-server-3

Routing example user sessions:

User ID      → Server
-----
user:1001     → web-server-3
user:1002     → web-server-1
user:1003     → web-server-1
user:1004     → web-server-3
user:1005     → web-server-3
user:1006     → web-server-1
user:1007     → web-server-3
user:1008     → web-server-1
user:1009     → web-server-2
user:1010     → web-server-3
user:1011     → web-server-2
user:1012     → web-server-1
user:1013     → web-server-1
user:1014     → web-server-1
user:1015     → web-server-2

Load distribution across 10,000 users:
web-server-3: 3267 [ ] 32.7%
web-server-1: 3521 [ ] 35.2%
web-server-2: 3212 [ ] 32.1%

✓ Each user consistently routes to same server (sticky sessions)
✓ Load balanced across all servers

DEMO 7: EDGE CASES & ROBUSTNESS
Handling various edge cases

Test 1: Empty ring
Get node for 'test': null (null expected)

Test 2: Single node
Get node for 'test': only-node
Get node for 'abc': only-node

Test 3: Keys with special characters
'key-with-dash' → only-node
'key_with_underscore' → only-node
'192.168.1.1' → only-node
'user@domain.com' → only-node
'👉' → only-node

Test 4: Large number of nodes
Added 100 nodes: 100 physical nodes, 15000 virtual nodes
Sample key routed to node: 65

✓ Handles edge cases gracefully

DEMO 8: PERFORMANCE ANALYSIS
Measurement of key lookup performance

Benchmarking key lookup performance:

Operations:      1,000,000 lookups
Total time:      464.47 ms
Avg per lookup:  0.464 μs
Throughput:      2,152,989 ops/sec

✓ Consistent hashing is very fast -  $O(\log n)$  complexity

Tue 7 Apr - 08:37 ~/UNI/adistr/consHashing/consistent-hashing / origin master ✓

```

Running the tests(without maven)

It is easier without maven that is why i decided to leave maven all together

```
32
33  ### 3. Run the Test Suite (27 tests)
34
35  `` bash
36  javac -cp target/classes -d target/classes src/main/java/org/ds/TestRunner.java
37  java -cp target/classes org.ds.TestRunner
38  ``
39
```

```
✓ testKeyRedistributionOnNodeAddition
✓ testKeyRedistributionOnNodeRemoval
✓ testRapidNodeAdditions
✓ testRapidNodeRemovals

└─ EDGE CASES & ROBUSTNESS TESTS ─┘
✓ testDuplicateNodeAddition
✓ testRemoveNonExistentNode
✓ testEmptyStringKey
✓ testVeryLongKey
✓ testSpecialCharacterKeys

└─ VIRTUAL NODES TESTS ─┘
✓ testVirtualNodeCount
✓ testVirtualNodesDistribution

└─ HASH FUNCTION TESTS ─┘
✓ testMD5HashConsistency
✓ testHashDiversity

└─ COMPLEX SCENARIO TESTS ─┘
✓ testCompleteLifecycle
✓ testCacheClusterScenario
✓ testLoadBalancingScenario

┌──────────────────────────────────┐
│ TEST SUMMARY                     │
└──────────────────────────────────┘

Total Tests: 27
Passed:      27 ✓
Failed:      0 ✗

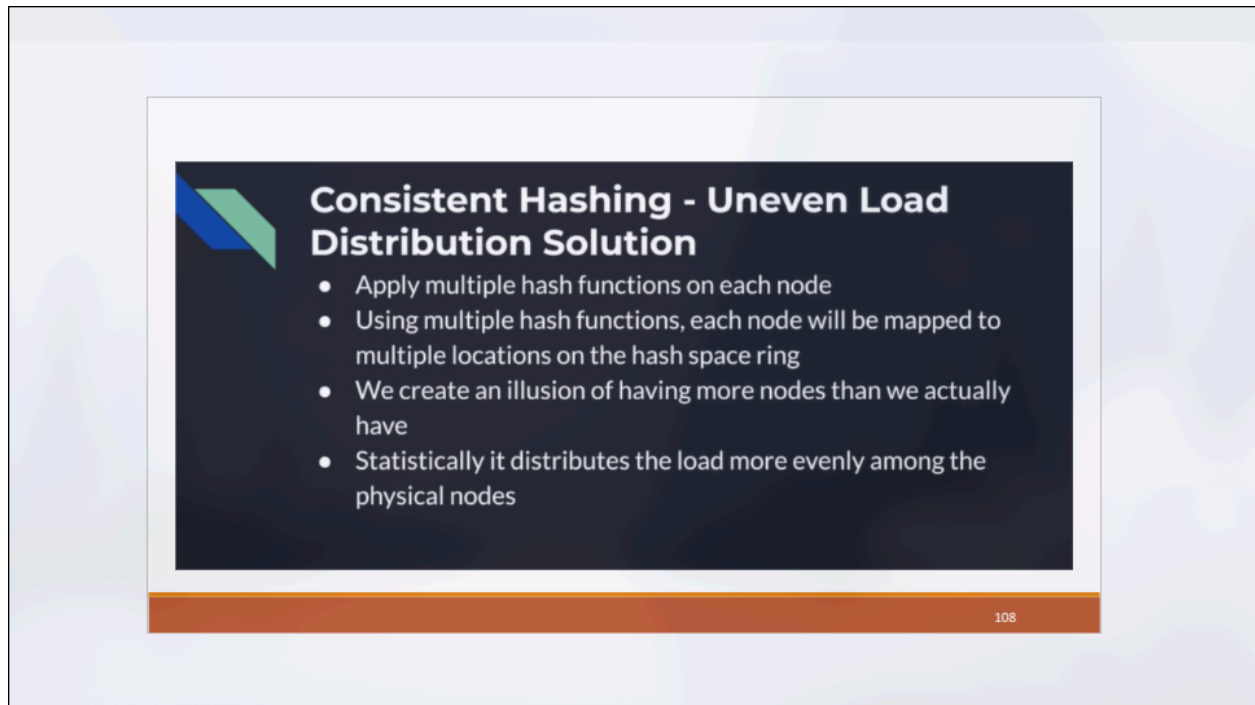
Success Rate: 100.0%

🎉 ALL TESTS PASSED!

Tue 7 Apr - 08:39 ~ /UNI/adistr/consHashing/consistent-hashing | origin @ master ✓
@arnull
```

As we can see they all have passed

Task 4(Adding new features) (solving the uneven distribution problem)



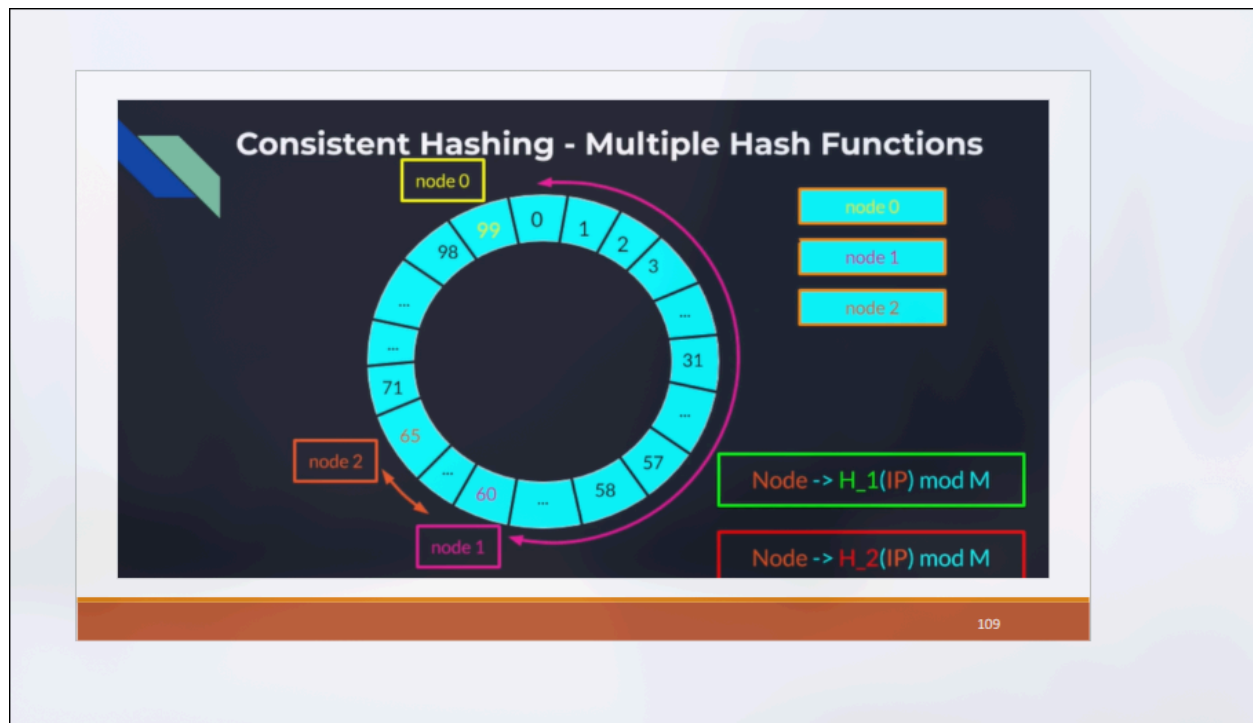
Consistent Hashing - Uneven Load Distribution Solution

- Apply multiple hash functions on each node
- Using multiple hash functions, each node will be mapped to multiple locations on the hash space ring
- We create an illusion of having more nodes than we actually have
- Statistically it distributes the load more evenly among the physical nodes

108

As we saw in the slides consistent hashing has a problem

Which uneven distribution (regular hashing method is better with even distributions)



The solution is present in the slide and that is multiple hashing rounds lets try to implement it

We did implement this with a new file
MultiAlgorithmConsistenHash.java

```
package org.ds;
```

```
import java.security.MessageDigest;

import java.nio.charset.StandardCharsets;

import java.util.*;

/**
 * Implementation of the "Multiple Hash Functions" solution from the slide.
 * Maps each node to the ring using distinct mathematical algorithms.
 */

public class MultiAlgorithmConsistentHash<T> extends ConsistentHash<T> {

    private final List<String> algorithms = Arrays.asList("MD5", "SHA-1",
"SHA-256");
```

```
public MultiAlgorithmConsistentHash(int virtualNodes) {  
  
    super(virtualNodes);  
  
}  
  
@Override  
  
public void addNode(T node) {  
  
    // As per the slide: Apply multiple hash functions on each node  
  
    for (int i = 0; i < virtualNodes; i++) {  
  
        // Cycle through algorithms (MD5, SHA1, SHA256) for each  
virtual node  
  
        String algo = algorithms.get(i % algorithms.size());  
  
        long hash = calculateHash(node.toString() + ":" + i, algo);  

```



```
// Put directly into the protected ring

ring.put(hash, node);

}

}

@Override

public void removeNode(T node) {

    for (int i = 0; i < virtualNodes; i++) {

        String algo = algorithms.get(i % algorithms.size());

        long hash = calculateHash(node.toString() + ":" + i, algo);

        ring.remove(hash);

    }

}
```

```
}

private long calculateHash(String key, String algorithm) {

    try {

        MessageDigest md = MessageDigest.getInstance(algorithm);

        byte[] bytes = md.digest(key.getBytes(StandardCharsets.UTF_8));

        long hash = 0;

        // Use 8 bytes for a 64-bit long hash

        for (int i = 0; i < Math.min(bytes.length, 8); i++) {

            hash = (hash << 8) | (bytes[i] & 0xFF);

        }

        return Math.abs(hash);
    }
}
```

```
        } catch (Exception e) {  
  
            throw new RuntimeException("Error calculating " + algorithm,  
e);  
  
        }  
  
    }  
  
}
```

This file override addnode and it uses three hashing algo (md5,sha1,sha256 aka sha2)

It still has one problem though the even distribution should also apply after node deletion and insertion as well we will ignore this for now

We added two test functions to test this

```
testMultiAlgorithmFunctionality()
```

```
testMultiAlgorithmBalance()
```

The first function is to test normal consistent hash function and they are not effected

And the second test is to make sure the mutli hash is really better for fairness after running the test again (after compiling of course we get the following results):

```
✓ testVirtualNodeCount
✓ testVirtualNodesDistribution

[ ] HASH FUNCTION TESTS
  ✓ testMD5HashConsistency
  ✓ testHashDiversity

[ ] COMPLEX SCENARIO TESTS
  ✓ testCompleteLifecycle
  ✓ testCacheClusterScenario
  ✓ testLoadBalancingScenario
  Testing Multiple Hash Function extension...
  ✓ testMultiAlgorithmFunctionality
  Testing load balance with multiple algorithms...
  ✓ testMultiAlgorithmBalance

TEST SUMMARY

Total Tests: 29
Passed: 29 ✓
Failed: 0 ✗

Success Rate: 100.0%

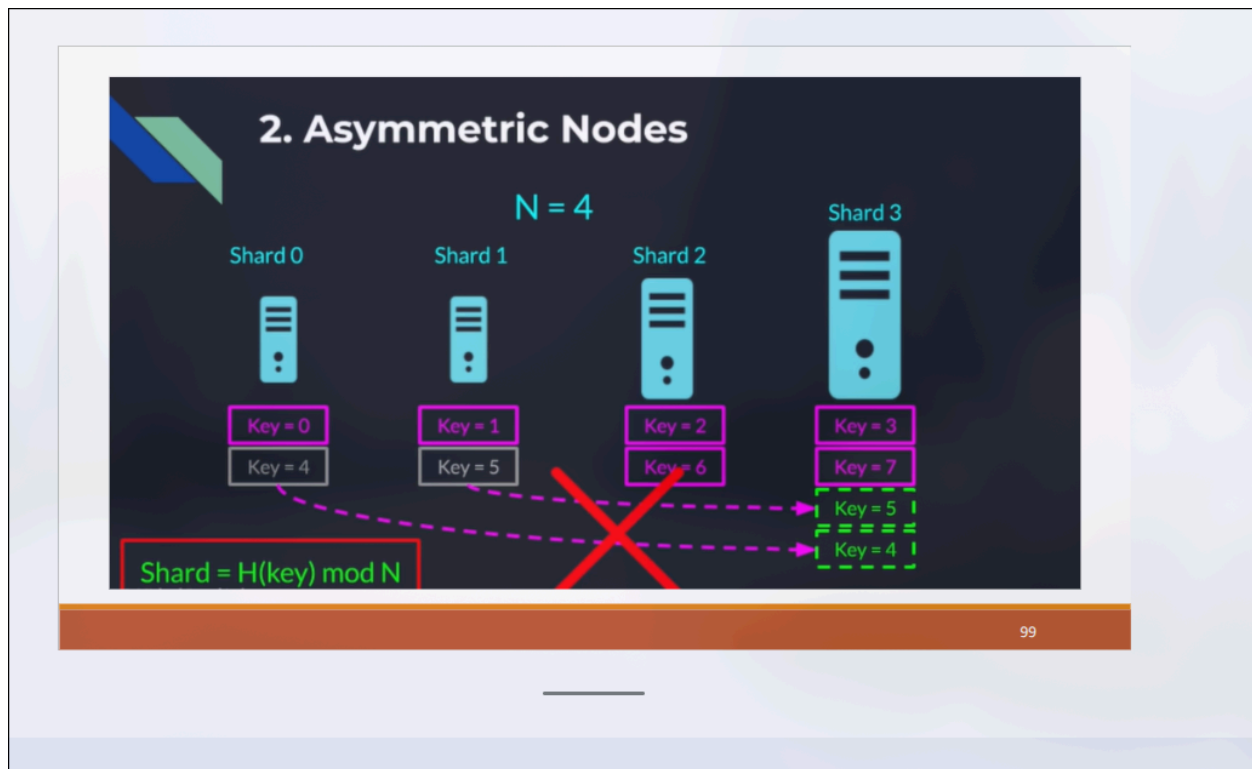
🎉 ALL TESTS PASSED!
```

Which is a good indication of our success

Task 4(critiquing the code) (code weaknesses)

1.Heterogeneous Node Weighting (Capacity-Aware Hashing):

The code doesn't account for asymmetric nodes and treat all nodes as equal



2. The Problem (The Hotspot Issue): Even with perfect hashing, a single "Hot Key" (like a viral tweet or a celebrity profile) can overwhelm one specific server. Consistent hashing normally forces 100% of that key's traffic to one node, even if that node is at 99% CPU and other nodes are idle.

3. the current code doesn't account for replication like at all with

Automatically erase High availability and Fault tolerance

Task 5 implementing a solution to the first weakness

We implement as a subclass using the following subclass

WeightedConsistentHash.java

```
"package org.ds;

import java.util.*;

/**
 * Extension: Implements "Heterogeneous Node Weighting".
 * Allows servers with more CPU/RAM to handle a larger share of the ring.
```

```
*/

public class WeightedConsistentHash<T> extends ConsistentHash<T> {

    // Store weights to ensure consistent removal

    private final Map<T, Integer> nodeWeights = new HashMap<>();

    public WeightedConsistentHash(int baseVirtualNodes) {

        super(baseVirtualNodes);

    }

    /**

     * Add a node with a specific capacity weight.

```



```
* @param node The physical node

* @param weight Capacity multiplier (e.g., 1 for small, 5 for large)

*/

public void addNode(T node, int weight) {

    if (weight <= 0) weight = 1;

    nodeWeights.put(node, weight);

    // The "Secret Sauce": Total Virtual Nodes = Base VN * Weight

    int totalVirtualNodesForThisNode = virtualNodes * weight;

    for (int i = 0; i < totalVirtualNodesForThisNode; i++) {

        long hash = hashFunction.hash(node.toString() + ":" + i);
```

```
        ring.put(hash, node);

    }

}

/**

 * Standard addNode defaults to weight 1

 */

@Override

public void addNode(T node) {

    addNode(node, 1);

}
```

```
@Override

public void removeNode(T node) {

    Integer weight = nodeWeights.getOrDefault(node, 1);

    int totalVirtualNodes = virtualNodes * weight;

    for (int i = 0; i < totalVirtualNodes; i++) {

        long hash = hashFunction.hash(node.toString() + ":" + i);

        ring.remove(hash);

    }

    nodeWeights.remove(node);

}
```

”

The previous code just associate a weight with each node

And add virtual nodes proportional to the weight

We add the following test to test the the ratio of weights

```
static void testWeightedDistribution() {  
  
    try {  
  
        System.out.println("\n└─ WEIGHTED CAPACITY TESTS  
└──────────────────────────────────┘");  
  
        // Base VN is 50  
  
        WeightedConsistentHash<String> weightedHash = new  
WeightedConsistentHash<>(50);
```

```
        System.out.println("Adding 'Small-Server' (Weight 1) and  
'Giant-Server' (Weight 10)...");  
  
        weightedHash.addNode("Small-Server", 1); // Gets 50 virtual  
nodes  
  
        weightedHash.addNode("Giant-Server", 10); // Gets 500 virtual  
nodes  
  
        // Run distribution check  
  
        int sampleSize = 10000;  
  
        Map<String, Integer> dist =  
weightedHash.getDistribution(sampleSize);  
  
        int smallCount = dist.get("Small-Server");  
  
        int giantCount = dist.get("Giant-Server");
```

```
double ratio = (double) giantCount / smallCount;

System.out.printf("  Small Server keys: %d\n", smallCount);

System.out.printf("  Giant Server keys: %d\n", giantCount);

System.out.printf("  Measured Ratio:    %.2fx (Target:
~10x)\n", ratio);

    // Verify Giant server took significantly more (at least 7x to
account for hash variance)

    assertTrue(ratio > 7.0, "Giant server should have significantly
more load");

    pass("testWeightedDistribution");

} catch (Exception e) {

    fail("testWeightedDistribution", e);
```

```
| WEIGHTED CAPACITY TESTS |  
Adding 'Small-Server' (Weight 1) and 'Giant-Server' (Weight 10)...  
  Small Server keys: 906  
  Giant Server keys: 9094  
  Measured Ratio:    10.04x (Target: ~10x)  
✓ testWeightedDistribution
```

TEST SUMMARY

Total Tests: 30
Passed: 30 ✓

After running the test our final test also passed because the distribution ratio is 10.04 greater than 7

This means that the giant server get 10 times the number of virtual nodes.