






# Consistent Hashing - 1 Hour Session

---

**Duration:** 60 minutes **Level:** Intermediate

## Session Agenda

---

-  Introduction to Consistent Hashing (10 min)
-  Hash Ring Fundamentals (15 min)
-  Virtual Nodes (10 min)
-  Rebalancing & Data Migration (10 min)
-  Implementation & Real-World Applications (15 min)

## Learning Objectives

---

By the end of this session, you will understand: - Why traditional hashing fails in distributed systems - How consistent hashing solves the redistribution problem - The hash ring concept and key placement - Virtual nodes and their benefits - How to handle node additions and removals - Real-world applications in distributed databases and caches

## 1. Introduction to Consistent Hashing (10 min)

---

### The Problem with Traditional Hashing

Simple Modulo Hashing:

```
server_index = hash(key) % number_of_servers
```

Example with 4 servers:

```
hash("user:123") = 12345
```

```
server = 12345 % 4 = 1 → Server 1
```

## What happens when we add/remove a server?

Before: 4 servers

```
hash("user:123") % 4 = 1 → Server 1
```

```
hash("user:456") % 4 = 0 → Server 0
```

```
hash("user:789") % 4 = 3 → Server 3
```


After: 5 servers (added one)

```
hash("user:123") % 5 = 0 → Server 0 (MOVED!)
```


```
hash("user:456") % 5 = 0 → Server 0 (same)
```

```
hash("user:789") % 5 = 4 → Server 4 (MOVED!)
```

Result: ~80% of keys need to be remapped! 💥

 **Warning** With N servers, adding or removing one server causes (N-1)/N keys to be remapped. For 100 servers, that's 99% of all data!

## Why This Matters

**Impact on Distributed Systems:** -  **Cache Invalidation:** Massive cache misses during scaling -  **Database Sharding:** Expensive data migration -  **Downtime:** System unavailable during rebalancing -  **Cost:** Network bandwidth and compute for data movement

## The Solution: Consistent Hashing

**Key Insight:** Instead of mapping keys to servers directly, map both keys AND servers to a fixed hash space (ring).

Traditional: `key → hash → server_index`

Consistent: `key → hash → position_on_ring → nearest_server`

**Benefits:** - Only  $K/N$  keys remapped when adding/removing nodes ( $K$  = total keys,  $N$  = nodes) - Minimal data movement during scaling - Predictable load distribution

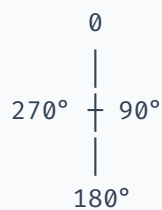
## 2. Hash Ring Fundamentals (15 min)

### The Hash Ring Concept

#### Creating the Ring:

Hash Space: 0 to  $2^{32} - 1$  (or any large number)

Visualized as a circle (ring)



`Position = hash(identifier) % ring_size`

## Placing Servers on the Ring

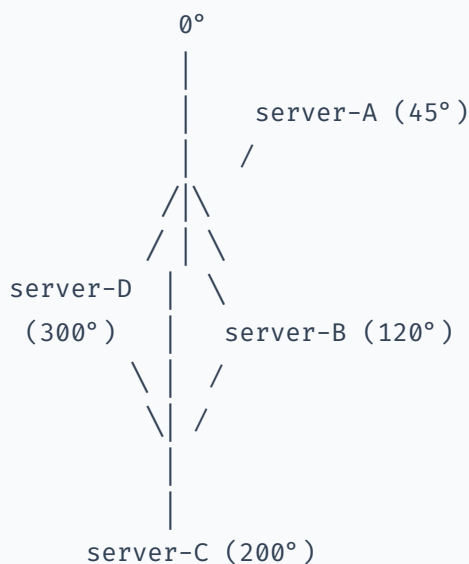
```
import hashlib

def hash_to_ring(key, ring_size=2**32):
    """Hash a key to a position on the ring"""
    hash_value = int(hashlib.md5(key.encode()).hexdigest(), 16)
    return hash_value % ring_size

# Place servers on the ring
servers = ["server-A", "server-B", "server-C", "server-D"]
server_positions = {s: hash_to_ring(s) for s in servers}

# Example positions (simplified to 0-360 for visualization)
# server-A: 45°
# server-B: 120°
# server-C: 200°
# server-D: 300°
```

### Visual Representation:



### Key Placement: Finding the Right Server

**Rule:** A key is assigned to the first server encountered when moving clockwise from the key's position.

```

def find_server(key, server_positions):
    """Find the server responsible for a key"""
    key_position = hash_to_ring(key)

    # Sort servers by position
    sorted_servers = sorted(server_positions.items(), key=lambda x: x[1])

    # Find first server clockwise from key position
    for server, position in sorted_servers:
        if position ≥ key_position:
            return server

    # Wrap around to first server
    return sorted_servers[0][0]

```

### Example:

Key "user:123" hashes to position 80°

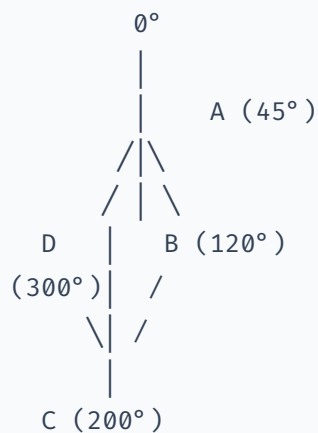
Ring positions:

- server-A: 45°
- server-B: 120° ← First server clockwise from 80°
- server-C: 200°
- server-D: 300°

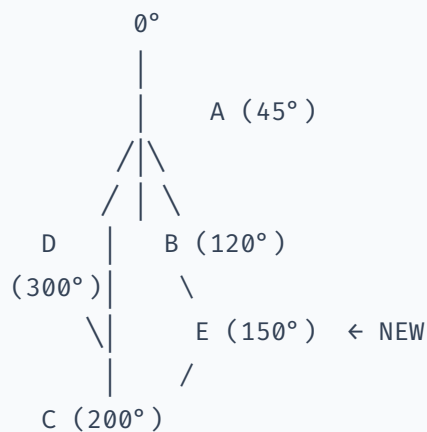
Result: "user:123" → server-B

## Adding a New Server

Before: 4 servers



After: Add server-E at 150°



Only keys between 120° and 150° move from C to E!

## Impact Analysis:

```
# Keys affected when adding server-E at 150°
# Only keys in range (120°, 150°] move from server-C to server-E

# Before: Keys 121°-200° → server-C
# After:  Keys 121°-150° → server-E
#         Keys 151°-200° → server-C

# Percentage moved: (150-120)/(360) ≈ 8.3% of total keys
# Much better than 80% with modulo hashing!
```

## Removing a Server

Remove server-B ( $120^\circ$ ):

Before:

- Keys  $46^\circ$ - $120^\circ \rightarrow$  server-B

After:

- Keys  $46^\circ$ - $120^\circ \rightarrow$  server-E (next clockwise server)

Only server-B's keys need to move!

## 3. Virtual Nodes (10 min)

---

### The Load Imbalance Problem

With few physical nodes, distribution can be uneven:

4 servers, random positions:

- server-A:  $10^\circ$  (covers  $10^\circ$  of ring = 2.8%)
- server-B:  $50^\circ$  (covers  $40^\circ$  of ring = 11.1%)
- server-C:  $300^\circ$  (covers  $250^\circ$  of ring = 69.4%)
- server-D:  $350^\circ$  (covers  $50^\circ$  of ring = 13.9%)

server-C handles 69% of all keys! 💥

### Solution: Virtual Nodes (VNodes)

**Idea:** Each physical server gets multiple positions on the ring.

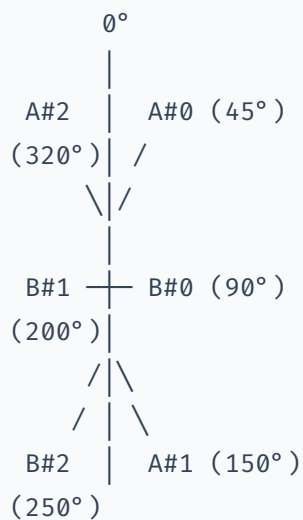
```
def create_virtual_nodes(server, num_vnodes=150):
    """Create multiple positions for a single server"""
    vnodes = {}
    for i in range(num_vnodes):
        vnode_key = f"{server}#vnode{i}"
        position = hash_to_ring(vnode_key)
        vnodes[position] = server
    return vnodes

# Example: server-A with 3 virtual nodes
# server-A#vnode0 → position 45°
# server-A#vnode1 → position 180°
# server-A#vnode2 → position 290°
```

## Visual with Virtual Nodes:

Physical servers: A, B  
Virtual nodes per server: 3

Ring:



Now load is distributed more evenly!



## Benefits of Virtual Nodes

ASPECT	WITHOUT VNODES	WITH VNODES
Load Distribution	Uneven	Even
Adding Node	Affects 1 range	Affects many small ranges
Heterogeneous Hardware	Not supported	Supported (more vnodes for powerful servers)
Failure Impact	Large chunk unavailable	Small chunks distributed

## Handling Heterogeneous Hardware

```
# More powerful servers get more virtual nodes
server_vnodes = {
    "server-A": 100,    # Standard server
    "server-B": 100,    # Standard server
    "server-C": 200,    # 2x capacity server
    "server-D": 50,     # Half capacity server
}

# server-C handles ~2x the load of A or B
# server-D handles ~0.5x the load of A or B
```

## Choosing the Number of Virtual Nodes

Factors to consider:

- More vnodes = better distribution, but more memory for ring
- Typical range: 100-200 vnodes per physical node
- Amazon DynamoDB uses 256 vnodes per node

Memory calculation:

- 100 physical nodes × 150 vnodes = 15,000 ring entries
- Each entry: ~100 bytes (position + server reference)
- Total: ~1.5 MB (negligible)

## 4. Rebalancing & Data Migration (10 min)

---

### When Rebalancing Occurs

1. **Node Addition:** New server joins the cluster
2. **Node Removal:** Server leaves (planned or failure)
3. **Capacity Change:** Adjusting virtual node count

### Rebalancing Strategy

#### Adding a Node:

```
def add_node(ring, new_server, num_vnodes=150):
    """Add a new server to the ring"""
    keys_to_migrate = {}

    for i in range(num_vnodes):
        vnode_key = f"{new_server}#vnode{i}"
        new_position = hash_to_ring(vnode_key)

        # Find the server that currently owns this position
        current_owner = find_server_at_position(ring, new_position)

        # Keys between previous position and new position move to new server
        prev_position = find_previous_position(ring, new_position)
        keys_to_migrate[new_position] = {
            'from': current_owner,
            'range': (prev_position, new_position)
        }

    # Add new vnode to ring
    ring[new_position] = new_server

    return keys_to_migrate
```

#### Migration Process:

Step 1: Add new node to ring (metadata only)  
Step 2: New node starts accepting writes for its range  
Step 3: Background migration of existing keys  
Step 4: Old node stops serving migrated keys

Timeline:

t=0 [Add to ring]  
t=1 [Accept new writes]  
t=2-10 [Background migration]  
t=11 [Complete]

## Handling Node Failures

### Graceful Removal:

```
def remove_node_graceful(ring, server):  
    """Gracefully remove a server"""  
    # 1. Stop accepting new writes  
    server.stop_writes()  
  
    # 2. Migrate data to successor nodes  
    for vnode_position in get_vnodes(ring, server):  
        successor = find_next_server(ring, vnode_position)  
        migrate_data(server, successor, vnode_position)  
  
    # 3. Remove from ring  
    remove_vnodes(ring, server)
```

### Sudden Failure:

When a node fails unexpectedly:

1. Detect failure (heartbeat timeout)
2. Remove from ring immediately
3. Successor nodes now responsible for failed node's range
4. If replication exists, replicas serve requests
5. Re-replicate data to maintain replication factor

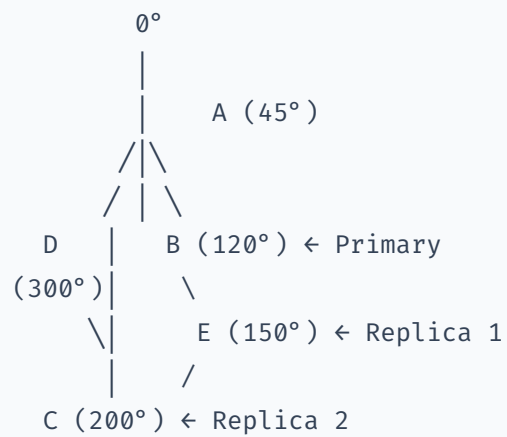
## Data Replication with Consistent Hashing

Replication Factor: 3

Strategy: Replicate to N-1 successor nodes

Key "user:123" at position 80°:

- Primary: server-B (120°)
- Replica 1: server-E (150°)
- Replica 2: server-C (200°)



## 5. Implementation & Real-World Applications (15 min)

---

### Complete Implementation

```

import hashlib
from bisect import bisect_right

class ConsistentHashRing:
    def __init__(self, num_vnodes=150):
        self.num_vnodes = num_vnodes
        self.ring = {} # position → server
        self.sorted_positions = []
        self.servers = set()

    def _hash(self, key):
        """Generate hash for a key"""
        return int(hashlib.md5(key.encode()).hexdigest(), 16)

    def add_server(self, server):
        """Add a server with virtual nodes"""
        self.servers.add(server)
        for i in range(self.num_vnodes):
            vnode_key = f"{server}#vnode{i}"
            position = self._hash(vnode_key)
            self.ring[position] = server

        self.sorted_positions = sorted(self.ring.keys())

    def remove_server(self, server):
        """Remove a server and its virtual nodes"""
        self.servers.discard(server)
        positions_to_remove = [
            pos for pos, srv in self.ring.items() if srv == server
        ]
        for pos in positions_to_remove:
            del self.ring[pos]

        self.sorted_positions = sorted(self.ring.keys())

    def get_server(self, key):
        """Find the server responsible for a key"""
        if not self.ring:
            return None

        key_hash = self._hash(key)

```

```

        # Find first position  $\geq$  key_hash
        idx = bisect_right(self.sorted_positions, key_hash)

        # Wrap around if necessary
        if idx == len(self.sorted_positions):
            idx = 0

        position = self.sorted_positions[idx]
        return self.ring[position]

    def get_servers_for_replication(self, key, replicas=3):
        """Get N servers for replication"""
        if not self.ring or replicas > len(self.servers):
            return list(self.servers)

        key_hash = self._hash(key)
        idx = bisect_right(self.sorted_positions, key_hash)

        servers = []
        seen = set()

        while len(servers) < replicas:
            if idx  $\geq$  len(self.sorted_positions):
                idx = 0

            server = self.ring[self.sorted_positions[idx]]
            if server not in seen:
                servers.append(server)
                seen.add(server)

            idx += 1

        return servers

# Usage
ring = ConsistentHashRing(num_vnodes=150)
ring.add_server("server-A")
ring.add_server("server-B")
ring.add_server("server-C")

# Find server for a key
server = ring.get_server("user:12345")
print(f"user:12345  $\rightarrow$  {server}")

```

```
# Get replicas
replicas = ring.get_servers_for_replication("user:12345", replicas=3)
print(f"Replicas: {replicas}")
```

## Real-World Applications

### 1. Amazon DynamoDB

DynamoDB Partitioning:

- Uses consistent hashing for partition key distribution
- 256 virtual nodes per physical partition
- Automatic rebalancing during scaling

Partition Key → Hash → Partition

"user:123" → 0x7F... → Partition-B

### 2. Apache Cassandra

Cassandra Token Ring:

- Each node owns a range of tokens
- Virtual nodes (vnodes) enabled by default
- Configurable replication factor

```
CREATE KEYSPACE my_keyspace
WITH replication = {
    'class': 'NetworkTopologyStrategy',
    'datacenter1': 3
};
```



### 3. Redis Cluster

Redis Cluster:

- 16384 hash slots (not continuous ring)
- Each node owns a subset of slots
- Manual or automatic slot migration

```
CLUSTER ADDSLOTS 0 1 2 3 ... 5460 # Node A
```

```
CLUSTER ADDSLOTS 5461 ... 10922 # Node B
```

```
CLUSTER ADDSLOTS 10923 ... 16383 # Node C
```

### 4. Content Delivery Networks (CDNs)

CDN Edge Server Selection:

- Hash(content\_url) → Edge server
- Ensures same content always served from same edge
- Maximizes cache hit rate

Request: GET /images/logo.png

Hash: 0x3F...

Edge: edge-server-tokyo-2

## Load Balancing with Consistent Hashing

```
class ConsistentHashLoadBalancer:
    def __init__(self):
        self.ring = ConsistentHashRing(num_vnodes=100)
        self.server_health = {}

    def add_backend(self, server, weight=1):
        """Add backend server with optional weight"""
        # More vnodes for higher weight
        vnodes = 100 * weight
        self.ring.num_vnodes = vnodes
        self.ring.add_server(server)
        self.server_health[server] = True

    def get_backend(self, request_key):
        """Get backend for a request"""
        # Use session ID or client IP as key for sticky sessions
        server = self.ring.get_server(request_key)

        # Health check
        if not self.server_health.get(server, False):
            # Fallback to next healthy server
            return self._get_next_healthy(request_key)

        return server

    def _get_next_healthy(self, key):
        """Find next healthy server on the ring"""
        servers = self.ring.get_servers_for_replication(key,
replicas=len(self.ring.servers))
        for server in servers:
            if self.server_health.get(server, False):
                return server
        return None
```

# Common Pitfalls & Solutions

---

## 1. Hot Spots

**Problem:** Some keys are accessed much more frequently.

```
Popular user "celebrity:123" → server-B  
Server-B gets 100x more traffic than others
```

**Solutions:** - Add read replicas for hot keys - Cache hot keys at application layer - Split hot keys across multiple entries

## 2. Uneven Virtual Node Distribution

**Problem:** Random hashing can still create clusters.

**Solution:** Use deterministic placement or jump consistent hashing.

```
# Jump Consistent Hash (Google)  
def jump_consistent_hash(key, num_buckets):  
    b, j = -1, 0  
    while j < num_buckets:  
        b = j  
        key = ((key * 2862933555777941757) + 1) & 0xFFFFFFFFFFFFFFFF  
        j = int((b + 1) * (float(1 << 31) / float((key >> 33) + 1)))  
    return b
```

## 3. Ring Metadata Synchronization

**Problem:** All nodes need consistent view of the ring.

**Solutions:** - Gossip protocol (Cassandra) - Centralized coordinator (ZooKeeper) - Consensus algorithm (Raft)

## 4. Data Migration During Rebalancing

**Problem:** Large data movement affects performance.

**Solutions:** - Background migration with rate limiting - Dual-write during transition - Lazy migration (migrate on access)



## Comparison with Alternatives

APPROACH	REDISTRIBUTION	COMPLEXITY	USE CASE
Modulo Hash	~100% on change	Low	Static clusters
Consistent Hash	~K/N on change	Medium	Dynamic clusters
Rendezvous Hash	~K/N on change	Medium	When order matters
Jump Hash	~K/N on change	Low	Memory-constrained



## Key Takeaways

1. **Consistent hashing minimizes data movement** when scaling clusters
2. **Virtual nodes ensure even distribution** and support heterogeneous hardware
3. **The hash ring is a fundamental building block** for distributed databases and caches
4. **Replication integrates naturally** by using successor nodes
5. **Real systems like DynamoDB, Cassandra, and Redis** all use variants of consistent hashing



## Practical Exercise

**Scenario:** Design a distributed cache for a social media platform.

**Requirements:** - 10 cache servers initially - Must scale to 50 servers - Session data must be sticky (same user → same server) - Handle server failures gracefully

**Questions:** 1. How many virtual nodes per server? 2. How to handle user session stickiness? 3. What happens when a server fails? 4. How to migrate data when adding servers?



## Quick Reference

---

### Hash Ring Operations

```
# Add server
ring.add_server("server-new")

# Remove server
ring.remove_server("server-old")

# Find server for key
server = ring.get_server("my-key")

# Get replicas
replicas = ring.get_servers_for_replication("my-key", n=3)
```

### Key Formulas

Keys remapped (modulo):  $(N-1)/N \approx 100\%$  for large  $N$   
Keys remapped (consistent):  $K/N$  where  $K$ =total keys,  $N$ =nodes

Virtual nodes memory:  $\text{nodes} \times \text{vnodes} \times \sim 100 \text{ bytes}$   
Recommended vnodes: 100-256 per physical node

**Session End - Thank You!** 🎉

#consistent-hashing #distributed-systems #partitioning #architecture #session-notes