Distributed Key-Value Store System Design (Like Redis, DynamoDB, Cassandra)

Table of Contents

Problem Statement & Requirements

Problem Definition

Design a distributed key-value store that can:

- Store billions of key-value pairs
- Handle millions of operations per second
- Provide high availability (99.99%+)
- Scale horizontally across thousands of nodes
- Support various consistency models
- Replicate data across multiple datacenters

Real-World Examples: Redis, Memcached, DynamoDB, Cassandra, etcd, Consul

---

Functional Requirements

1. **Basic Operations**
    - `PUT(key, value)`: Store a key-value pair
    - `GET(key)`: Retrieve value by key
    - `DELETE(key)`: Remove a key-value pair
    - `EXISTS(key)`: Check if key exists
2. **Advanced Operations**
    - `LIST(prefix)`: List keys by prefix
    - `SCAN(cursor)`: Iterate over keys

- `TTL(key, seconds)`: Set time-to-live
- `BATCH_PUT(keys, values)`: Atomic batch write
- `COMPARE_AND_SWAP(key, old_value, new_value)`: Conditional update

3. **Configuration**
    - Configurable replication factor
    - Tunable consistency levels
    - Custom data retention policies
    - Multi-tenancy support

Non-Functional Requirements

1. **Performance**
    - Read latency: P99 < 10ms
    - Write latency: P99 < 20ms
    - Throughput: 100K+ ops/second per node
    - Support 10M+ ops/second cluster-wide
2. **Scalability**
    - Horizontal scaling (add/remove nodes)
    - Support 1000+ node clusters
    - Store petabytes of data
    - Linear scalability with node count
3. **Availability**
    - 99.99% uptime (52 minutes downtime/year)
    - No single point of failure
    - Automatic failover (< 30 seconds)
    - Graceful degradation
4. **Consistency**
    - Support multiple consistency models:
        - Strong consistency
        - Eventual consistency
        - Read-your-writes consistency
    - Tunable per operation
5. **Durability**
    - Data persisted to disk
    - Replication across nodes/datacenters
    - Snapshot and backup support
    - 99.999999999% durability (11 nines)

6. **Partition Tolerance**
    - Continue operating during network partitions
    - Automatic partition healing
    - Data reconciliation

Capacity Estimation

Assumptions

```
Total Data: 100 TB
Average Key Size: 50 bytes
Average Value Size: 1 KB
Total Keys: 100 TB / 1 KB ≈ 100 Billion keys
Replication Factor: 3
Actual Storage: 100 TB × 3 = 300 TB
```

QPS (Queries Per Second)

```
Read:Write Ratio: 80:20 (read-heavy workload)
Total Operations: 10 Million ops/second

Reads: 8M ops/sec
Writes: 2M ops/sec

Per Node (100 nodes):
- Reads: 80K ops/sec
- Writes: 20K ops/sec
```

Memory Requirements

```
Metadata per key: 100 bytes (key, value pointer, timestamps, version)
Total Keys: 100B

Memory for indexes:
100B keys × 100 bytes = 10 TB metadata

With caching (20% hot data):
Cache: 20 TB of actual data + 2 TB metadata = 22 TB
Per Node (100 nodes): 220 GB memory
```

Network Bandwidth

```
Average Operation Size: 1 KB
10M ops/sec × 1 KB = 10 GB/sec

With replication (factor 3):
Writes: 2M × 3 replicas = 6M ops/sec
Total Bandwidth: 16 GB/sec

Per Node: 160 MB/sec
```

Storage Requirements

```
Data: 300 TB (with 3x replication)
Write-Ahead Log (WAL): 30 TB (10% overhead)
Snapshots: 100 TB (1 full snapshot)
Total: 430 TB

Per Node (100 nodes): 4.3 TB per node
Recommended: 8 TB SSD per node (room for growth)
```

API Design

Basic Operations

```
# Put - Store a key-value pair
PUT /api/v1/keys/{key}
Headers:
  Content-Type: application/json
  Consistency-Level: QUORUM (optional)
Body: {
  "value": "some_value",
  "ttl": 3600,  // optional, in seconds
  "if_not_exists": false  // optional, for conditional put
}
Response: {
  "key": "user:123",
  "version": 5,
  "timestamp": 1640995200,
  "status": "success"
}

# Get - Retrieve value by key
GET /api/v1/keys/{key}
Headers:
  Consistency-Level: EVENTUAL (optional)
Response: {
  "key": "user:123",
```

```
  "value": "some_value",
  "version": 5,
  "timestamp": 1640995200,
  "ttl_remaining": 3500
}

# Delete - Remove a key
DELETE /api/v1/keys/{key}
Response: {
  "key": "user:123",
  "deleted": true,
  "version": 6
}

# Exists - Check if key exists
HEAD /api/v1/keys/{key}
Response:
  200 OK (exists)
  404 Not Found (doesn't exist)
```

Batch Operations

```
# Batch Put - Write multiple keys atomically
POST /api/v1/batch/put
Body: {
  "operations": [
    {"key": "user:123", "value": "data1"},
    {"key": "user:456", "value": "data2"}
  ],
  "atomic": true  // fail all if any fails
}
Response: {
  "success_count": 2,
  "failed_keys": [],
  "results": [...]
}

# Batch Get - Read multiple keys
POST /api/v1/batch/get
Body: {
  "keys": ["user:123", "user:456", "user:789"]
}
Response: {
  "results": [
    {"key": "user:123", "value": "data1", "found": true},
    {"key": "user:456", "value": "data2", "found": true},
    {"key": "user:789", "found": false}
  ]
}
```

Advanced Operations

```
# Compare and Swap - Atomic conditional update
POST /api/v1/keys/{key}/cas
Body: {
  "expected_version": 5,
  "new_value": "updated_value"
}
Response: {
  "success": true,
  "new_version": 6
}

# List Keys by Prefix
GET /api/v1/keys?prefix=user:&limit=100
Response: {
  "keys": ["user:123", "user:456", ...],
  "cursor": "next_page_token",
  "has_more": true
}

# Scan - Iterate over all keys
GET /api/v1/scan?cursor={token}&count=1000
Response: {
  "keys": [...],
  "next_cursor": "new_token",
  "done": false
}

# Set TTL - Time to live
POST /api/v1/keys/{key}/ttl
Body: {
  "ttl_seconds": 3600
}

# Increment - Atomic increment (for counters)
POST /api/v1/keys/{key}/incr
Body: {
  "delta": 1
}
Response: {
  "key": "counter:views",
  "value": 1001,
  "version": 1002
}
```

Admin Operations

```
# Get Cluster Status
GET /api/v1/admin/cluster/status
Response: {
  "nodes": [
    {
      "node_id": "node-1",
      "address": "10.0.1.5:6379",
      "status": "healthy",
      "role": "master",
      "keys_count": 10000000,
      "memory_used_mb": 5120
    }
  ],
  "total_keys": 100000000,
  "replication_factor": 3
}

# Add Node to Cluster
POST /api/v1/admin/cluster/nodes
Body: {
  "node_address": "10.0.1.10:6379"
}

# Rebalance Cluster
POST /api/v1/admin/cluster/rebalance
```

Data Model & Storage

In-Memory Index Structure

```
Hash Table (Main Index):
┌─────────────────────────────────────────────────┐
│ Key Hash → Value Pointer                        │
├─────────────────────────────────────────────────┤
│ hash("user:123") → 0x7FA2B4C0                   │
│ hash("user:456") → 0x7FA2B800                   │
│ hash("session:abc") → 0x7FA2BC00                │
└─────────────────────────────────────────────────┘

Value Pointer Structure:
┌─────────────────────────────────────────────────┐
│ struct ValuePointer {                           │
│   char* data;          // pointer to value      │
│   uint32_t size;       // value size            │
│   uint64_t version;    // version number        │
│   uint64_t timestamp;  // last modified         │
│   uint32_t ttl;        // time to live          │
│   uint8_t flags;       // metadata flags        │
```

```
| }                                                    |
└──────────────────────────────────────────────────────┘
```

On-Disk Storage Format (SSTable)

```
SSTable (Sorted String Table) Layout:
┌──────────────────────────────────────────────────────┐
│                  Index Block                         │
│   ┌──────────────────────────────────────────┐       │
│   │ Key1 → Offset1                           │       │
│   │ Key2 → Offset2                           │       │
│   │ ...                                      │       │
│   └──────────────────────────────────────────┘       │
├──────────────────────────────────────────────────────┤
│                  Data Block 1                        │
│   ┌──────────────────────────────────────────┐       │
│   │ Key1 | Value1 | Metadata1                │       │
│   │ Key2 | Value2 | Metadata2                │       │
│   └──────────────────────────────────────────┘       │
├──────────────────────────────────────────────────────┤
│                  Data Block 2                        │
│   ┌──────────────────────────────────────────┐       │
│   │ Key3 | Value3 | Metadata3                │       │
│   └──────────────────────────────────────────┘       │
├──────────────────────────────────────────────────────┤
│                  Footer                              │
│   ┌──────────────────────────────────────────┐       │
│   │ Index Offset                             │       │
│   │ Bloom Filter                             │       │
│   │ Checksum                                 │       │
│   └──────────────────────────────────────────┘       │
└──────────────────────────────────────────────────────┘

Bloom Filter: Probabilistic data structure
- Check if key exists (no false negatives)
- ~10 bytes per key
- 1% false positive rate
```

LSM Tree (Log-Structured Merge Tree)

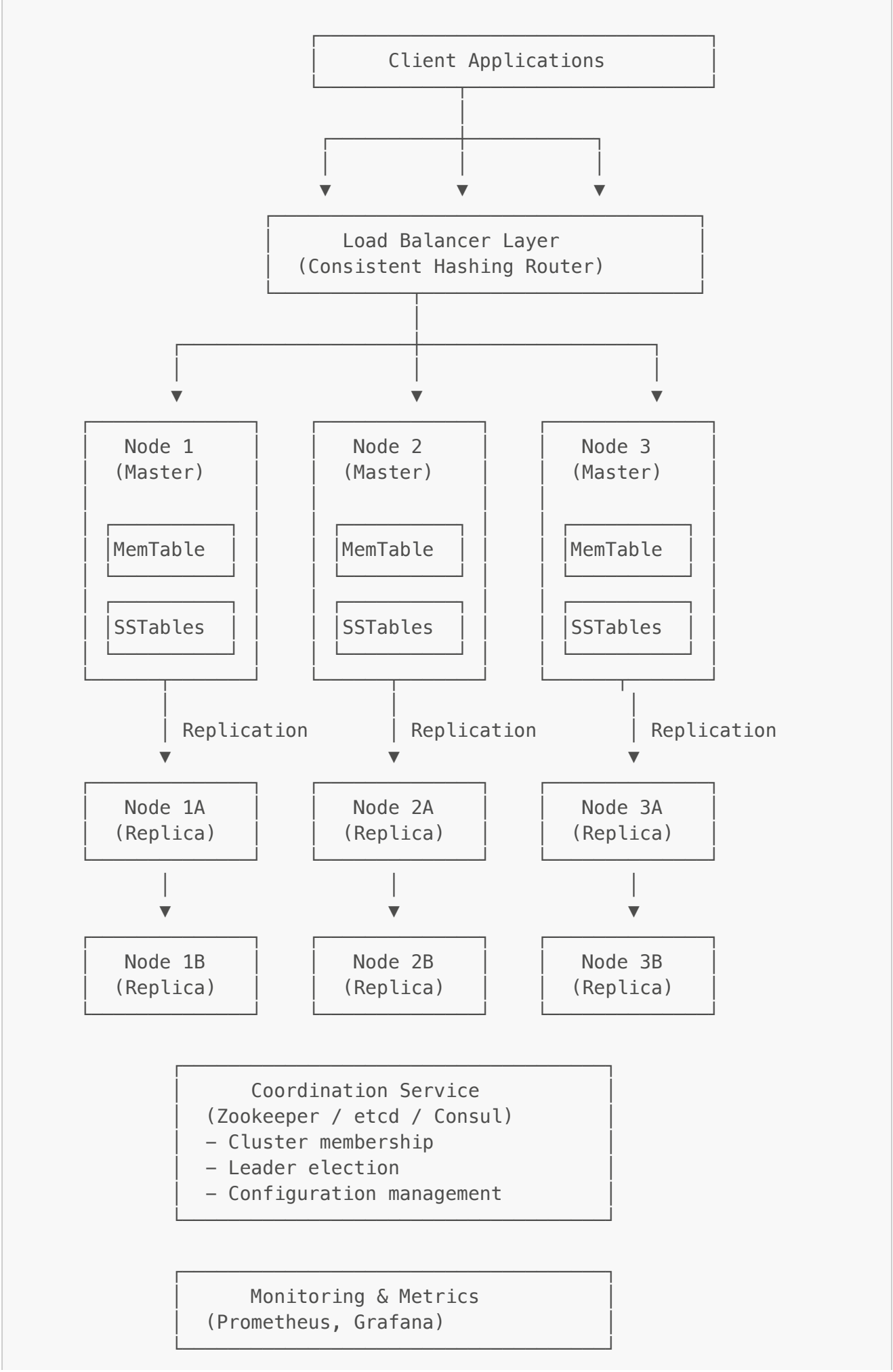
```
Write Path (LSM Tree):

1. MemTable (In-Memory):
    ┌──────────────────┐
    │  Sorted Map      │  ← New writes go here
    │  (Red-Black Tree)│
    └──────────────────┘
            │
```

```
            │ When full (64MB)
            ▼
2. Immutable MemTable:
    ┌──────────────────────┐
    │  Frozen MemTable     │  ← No more writes
    └──────────────────────┘
            │
            │ Flush to disk
            ▼
3. SSTable Level 0:
    ┌──────────────────────┐
    │  SSTable Files       │  ← Sorted on disk
    │  (may overlap)       │
    └──────────────────────┘
            │
            │ Compaction
            ▼
4. SSTable Level 1+:
    ┌──────────────────────┐
    │  Compacted           │  ← Merged, sorted
    │  SSTable Files       │     (no overlap)
    └──────────────────────┘

Compaction Strategy:
- Size-Tiered: Merge SSTables of similar size
- Leveled: Each level 10x larger than previous
- Time-Window: Merge based on time ranges
```

Write-Ahead Log (WAL)

```
WAL Structure:
┌──────────────────────────────────────────────┐
│ Entry 1: PUT user:123 "value1" v1 ts1        │
│ Entry 2: PUT user:456 "value2" v1 ts2        │
│ Entry 3: DELETE user:789 v2 ts3              │
│ Entry 4: PUT session:abc "data" v1 ts4       │
│ ...                                          │
│ Checksum                                     │
└──────────────────────────────────────────────┘

Purpose:
- Durability (survive crashes)
- Replay on recovery
- Asynchronous flushing to disk
```

High-Level Architecture


Client Applications
Load Balancer Layer
(Consistent Hashing Router)
Node 1
(Master)
MemTable
SSTables
Node 2
(Master)
MemTable
SSTables
Node 3
(Master)
MemTable
SSTables
Replication
Replication
Replication
Node 1A
(Replica)
Node 2A
(Replica)
Node 3A
(Replica)
Node 1B
(Replica)
Node 2B
(Replica)
Node 3B
(Replica)
Coordination Service
(Zookeeper / etcd / Consul)
- Cluster membership
- Leader election
- Configuration management
Monitoring & Metrics
(Prometheus, Grafana)

Architecture Layers

1. **Client Layer**
    - Client libraries (smart clients)
    - Connection pooling
    - Automatic retry logic
    - Consistent hashing
2. **Routing Layer**
    - Load balancers
    - Request routing based on key
    - Health checks
    - Circuit breakers
3. **Storage Node Layer**
    - Master nodes (handle reads/writes)
    - Replica nodes (handle reads)
    - Data persistence
    - Replication coordination
4. **Coordination Layer**
    - Cluster membership
    - Leader election
    - Configuration sync
    - Failure detection
5. **Monitoring Layer**
    - Metrics collection
    - Alerting
    - Dashboard
    - Distributed tracing

Core Components

1. Storage Engine

Responsibilities:

- Store key-value pairs on disk
- Provide efficient reads/writes
- Handle compaction
- Manage cache

Implementation: LSM Tree

```
Components:

1. MemTable (In-Memory):
   - Sorted data structure (Skip List or Red-Black Tree)
   - Receives all writes
   - Fast writes (O(log n))
   - Limited size (64-128 MB)

2. Immutable MemTable:
   - Frozen when MemTable full
   - Background flush to disk
   - Read-only during flush

3. SSTable Files:
   - Immutable on-disk files
   - Sorted by key
   - Multiple levels (L0, L1, L2, ...)
   - Bloom filters for fast lookups

4. Write-Ahead Log (WAL):
   - Append-only log
   - Durability guarantee
   - Replayed on crash recovery

Operations:

Write Flow:
1. Append to WAL (sync to disk)
2. Write to MemTable
3. Return success
4. Background: Flush MemTable → SSTable

Read Flow:
1. Check MemTable
2. Check Immutable MemTable
3. Check SSTables (L0 → L1 → L2...)
4. Use Bloom filters to skip files
5. Return result or NOT_FOUND

Compaction:
- Merge overlapping SSTables
- Remove deleted keys
- Reduce read amplification
- Reclaim disk space
```

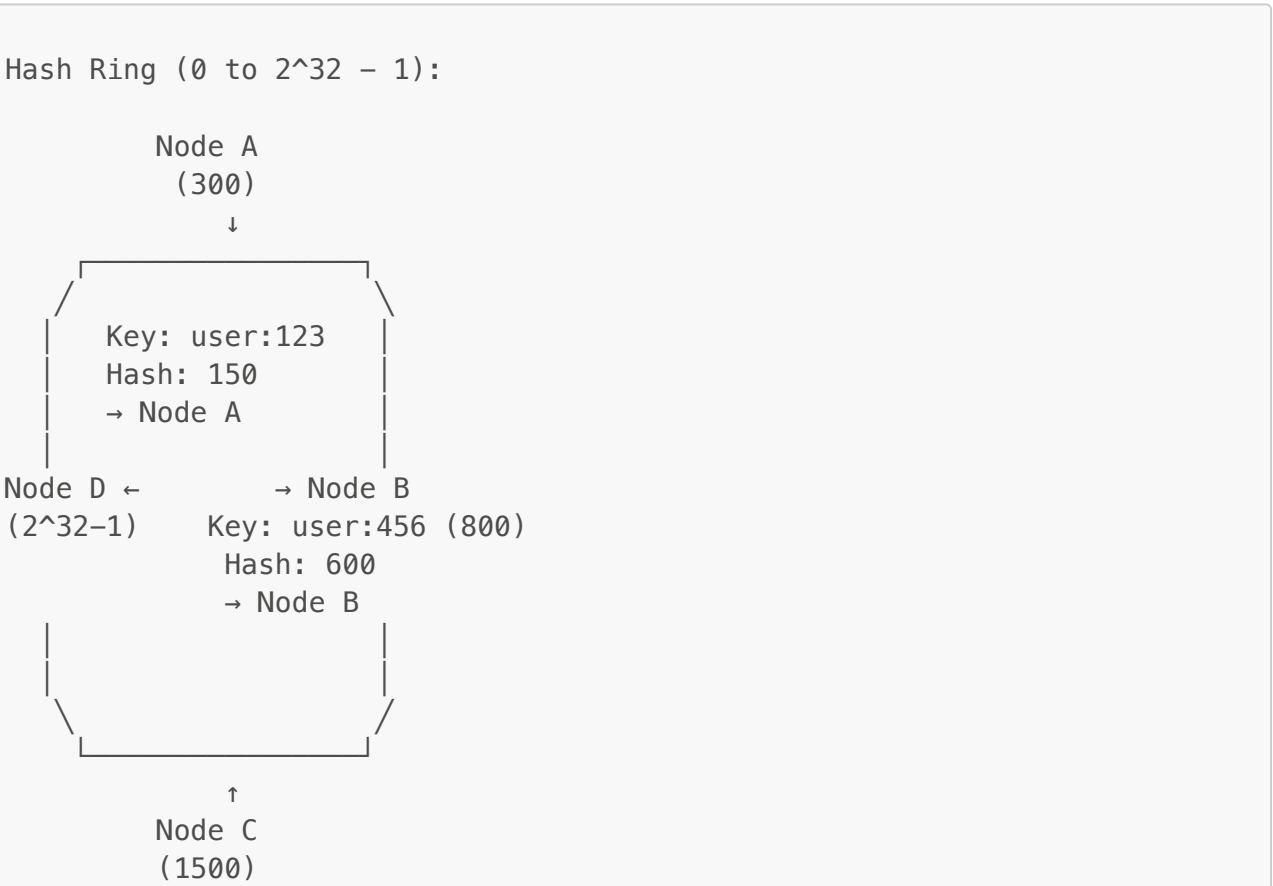
Alternative: B-Tree

```
Pros:
✓ Better read performance (lower read amplification)
✓ Simpler implementation
✓ In-place updates

Cons:
✗ Worse write performance (random I/O)
✗ Write amplification
✗ Fragmentation issues

Use When:
- Read-heavy workload
- Limited write throughput
- Strong consistency required
```

---

2. Consistent Hashing

Purpose: Distribute keys evenly across nodes




```
Algorithm:
1. Hash each node (node_id) → position on ring
2. Hash each key → position on ring
3. Walk clockwise to find first node
4. Key stored on that node (+ replicas)

Virtual Nodes:
```

```
- Each physical node → multiple virtual nodes
- Better load distribution
- Easier rebalancing
- Example: 256 virtual nodes per physical node
```

Key Operations:

```python
class ConsistentHash:
    def __init__(self, nodes, virtual_nodes=256):
        self.virtual_nodes = virtual_nodes
        self.ring = {}
        self.sorted_keys = []

        for node in nodes:
            self.add_node(node)

    def add_node(self, node):
        for i in range(self.virtual_nodes):
            virtual_key = f"{node}:{i}"
            hash_value = hash(virtual_key) % (2**32)
            self.ring[hash_value] = node

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

    def get_node(self, key):
        if not self.ring:
            return None

        hash_value = hash(key) % (2**32)

        # Binary search for first node >= hash_value
        idx = bisect.bisect_left(self.sorted_keys, hash_value)

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

        return self.ring[self.sorted_keys[idx]]

    def get_replicas(self, key, replication_factor=3):
        replicas = []
        hash_value = hash(key) % (2**32)
        idx = bisect.bisect_left(self.sorted_keys, hash_value)

        # Get next N unique nodes
        seen_nodes = set()
        while len(seen_nodes) < replication_factor:
            if idx >= len(self.sorted_keys):
                idx = 0

            node = self.ring[self.sorted_keys[idx]]
```

```
            if node not in seen_nodes:
                seen_nodes.add(node)
                replicas.append(node)

            idx += 1

        return replicas
```

Benefits:

- Minimal data movement when nodes added/removed
- Even distribution with virtual nodes
- Automatic load balancing
- Fault tolerance (replicas on different nodes)

---

3. Replication Coordinator

Responsibilities:

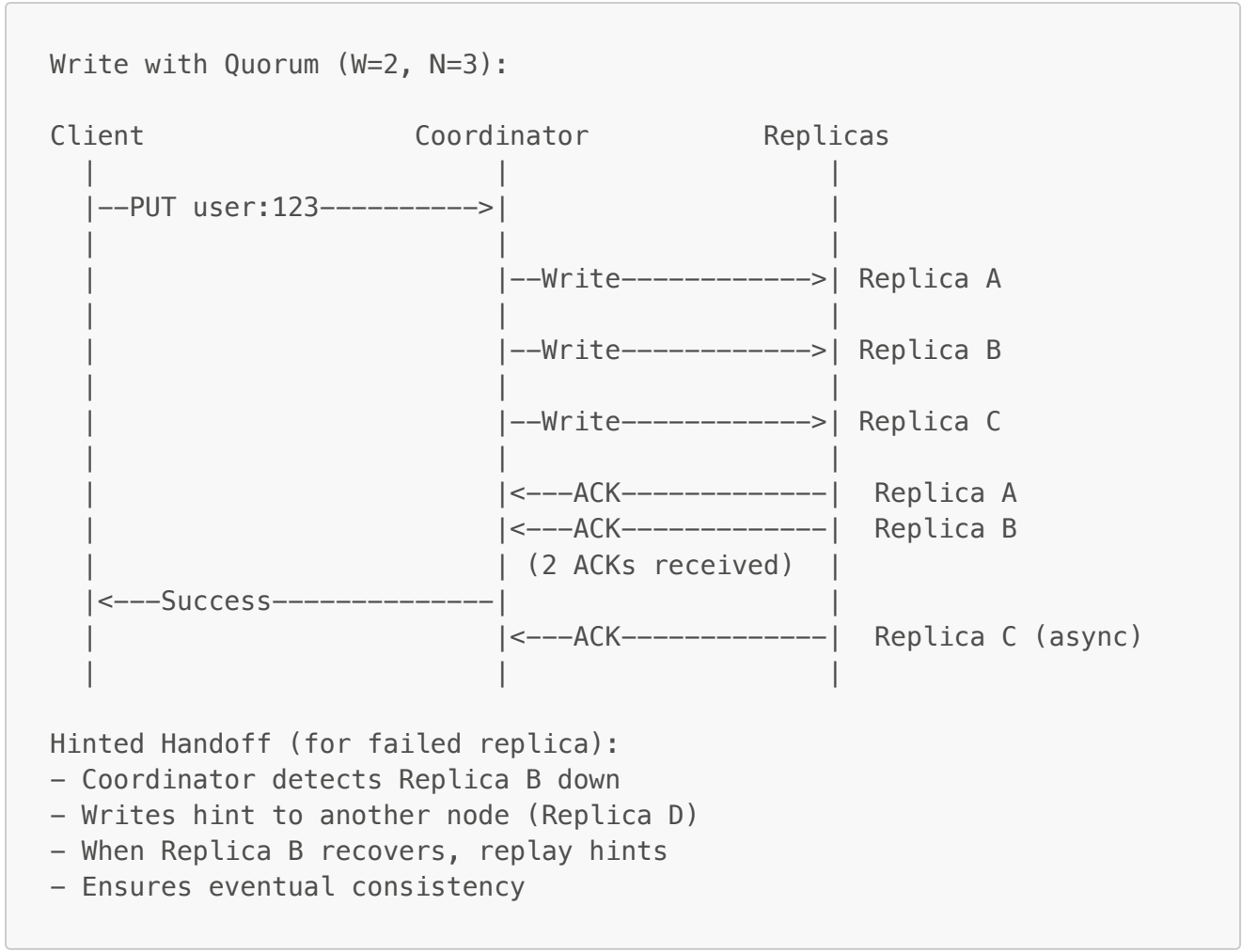
- Coordinate writes to replicas
- Handle consistency levels
- Detect and repair inconsistencies
- Manage replica placement

Replication Strategies:

```
1. Master-Slave Replication:

   Client → Master (Write)
   Master → Replicas (Async)
   Client → Replicas (Read)

   Pros: Simple, read scalability
   Cons: Write bottleneck, replication lag

2. Multi-Master Replication:

   Client → Any Node (Write)
   Node → Other Nodes (Async)

   Pros: Write scalability, no bottleneck
   Cons: Conflict resolution needed

3. Quorum-Based Replication:

   R + W > N (R=reads, W=writes, N=replicas)

   Example: N=3, W=2, R=2
   - Write to 2 out of 3 replicas
```

```
   - Read from 2 out of 3 replicas
   - Guaranteed to see latest write

   Pros: Tunable consistency
   Cons: Higher latency
```

Replication Protocol:



---

4. Gossip Protocol

Purpose: Disseminate cluster state without central coordinator

```
Gossip Protocol Flow:

Every node periodically (1 second):
1. Select random peer
2. Exchange state information
3. Merge states
4. Detect failures

Example:
Node A knows: {A:alive, B:alive, C:unknown}
```

```
Node B knows: {A:alive, B:alive, C:dead}

After gossip:
Node A knows: {A:alive, B:alive, C:dead}
Node B knows: {A:alive, B:alive, C:dead}

Information spreads exponentially:
- After log(N) rounds, all nodes know
- Self-healing on network partition
- No single point of failure
```

Implementation:

```python
class GossipProtocol:
    def __init__(self, node_id, peers):
        self.node_id = node_id
        self.peers = peers
        self.state = {node_id: {'status': 'alive', 'version': 0}}

    def gossip_round(self):
        # Select random peer
        peer = random.choice(self.peers)

        # Send state to peer
        peer_state = self.send_state(peer, self.state)

        # Merge states
        self.merge_state(peer_state)

        # Increment local version
        self.state[self.node_id]['version'] += 1

    def merge_state(self, peer_state):
        for node, info in peer_state.items():
            if node not in self.state:
                self.state[node] = info
            elif info['version'] > self.state[node]['version']:
                self.state[node] = info

    def detect_failures(self):
        current_time = time.time()
        for node, info in self.state.items():
            if current_time - info['last_seen'] > 10:  # 10 sec timeout
                info['status'] = 'suspected'
            if current_time - info['last_seen'] > 30:  # 30 sec timeout
                info['status'] = 'dead'
```

---

5. Failure Detector

Responsibilities:

- Detect node failures
- Trigger automatic failover
- Prevent false positives (network delays)

Phi Accrual Failure Detector:

```
Concept:
- Track heartbeat intervals
- Calculate probability of failure (phi)
- Threshold-based decision

Algorithm:
1. Track last N heartbeat arrival times
2. Calculate mean and variance
3. Compute phi value:
   φ(t) = -log10(P(arrival_time > t))

4. Decision:
   φ < 5: Node likely alive
   φ > 8: Node likely dead
   5 < φ < 8: Suspicious

Advantages:
- Adaptive to network conditions
- Configurable threshold
- Low false positive rate
```

---

6. Anti-Entropy (Merkle Trees)

Purpose: Detect and repair inconsistencies between replicas

```
Merkle Tree Structure:

                  Root Hash
                 (H1234567)
                /          \
            H1234          H5678
           /    \          /    \
        H12    H34      H56    H78
        / \    / \      / \    / \
      H1 H2  H3 H4    H5 H6  H7 H8
      |  |   |  |     |  |   |  |
     K1 K2  K3 K4    K5 K6  K7 K8

Each leaf = hash of key-value pair
Each parent = hash of children
```

```
Comparison Algorithm:
1. Exchange root hashes
2. If equal → replicas in sync
3. If different → recursively compare children
4. Identify differing keys
5. Sync only differing keys

Benefit: O(log n) comparisons instead of O(n)
```

Anti-Entropy Process:

```
Periodic background task (every hour):
1. Build Merkle tree for key range
2. Exchange tree with replica
3. Compare trees recursively
4. Identify inconsistent keys
5. Repair inconsistencies (read repair)

Benefits:
- Proactive inconsistency detection
- Efficient comparison (O(log n))
- Background operation (no latency impact)
- Self-healing system
```

Data Partitioning (Sharding)

Why Partition?

Problem: Single node limitations

- Storage: Limited disk space (single server ~10TB)
- Memory: Limited RAM for indexes
- CPU: Limited processing power
- Network: Single NIC bottleneck

Solution: Distribute data across multiple nodes

Partitioning Strategies

1. Hash-Based Partitioning

```
Partition = hash(key) % num_partitions

Example:
hash("user:123") = 8472
```

```
8472 % 10 = 2 → Partition 2

Pros:
✓ Even distribution
✓ Simple implementation
✓ Good for random access

Cons:
✗ Poor range queries
✗ All partitions needed for scan
```

2. Range-Based Partitioning

```
Partitions by key ranges:

Partition 1: A–F
Partition 2: G–M
Partition 3: N–S
Partition 4: T–Z

Example:
"user:alice" → Partition 1
"user:john" → Partition 2
"user:tom" → Partition 4

Pros:
✓ Efficient range queries
✓ Prefix searches fast
✓ Good for time-series data

Cons:
✗ Hot spots (uneven distribution)
✗ Rebalancing complex
```

3. Consistent Hashing (Recommended)

Already covered in Core Components section

Partition Management

Token Assignment:

```
Each node owns token ranges:

Node 1: tokens 0 – 99
Node 2: tokens 100 – 199
Node 3: tokens 200 – 299
```

```
...

Token = hash(key) % total_tokens

Benefits:
- Fine-grained control
- Easy rebalancing (move tokens)
- Flexible scaling
```

Rebalancing Strategy:

```
Add New Node:

Before:
Node A: [0-199]
Node B: [200-399]
Node C: [400-599]

Add Node D:

After:
Node A: [0-149]
Node B: [200-349]
Node C: [400-549]
Node D: [150-199, 350-399, 550-599]

Process:
1. Allocate tokens to new node
2. Stream data from existing nodes
3. Verify checksums
4. Update routing table
5. Drain old data
```

Replication Strategy

Replication Factor

```
N = Replication Factor (typically 3)

Each key stored on N nodes:
- Primary node (coordinator)
- N-1 replica nodes

Example: N=3
Key "user:123" → Node A (primary)
              → Node B (replica)
              → Node C (replica)
```

Replica Placement

Strategy: Cross-Rack and Cross-DC

```
Rack-Aware Placement:

Datacenter 1:
  Rack 1: Node A (primary)
  Rack 2: Node B (replica)

Datacenter 2:
  Rack 3: Node C (replica)

Benefits:
- Survive rack failure
- Survive datacenter failure
- Local reads in each DC
```

Replication Protocols

1. Synchronous Replication

```
Write Flow:
1. Client → Coordinator
2. Coordinator → All replicas (parallel)
3. Wait for all ACKs
4. Coordinator → Client (success)

Pros:
✓ Strong consistency
✓ No data loss

Cons:
✗ High latency (slowest replica)
✗ Reduced availability (any failure blocks)
```

2. Asynchronous Replication

```
Write Flow:
1. Client → Primary
2. Primary → Client (success immediately)
3. Primary → Replicas (background)

Pros:
✓ Low latency
```

```
✓ High availability

Cons:
✗ Eventual consistency
✗ Potential data loss
```

3. Quorum-Based (Hybrid)

```
Tunable Consistency:

N = 3 (total replicas)
W = 2 (write quorum)
R = 2 (read quorum)

Rule: R + W > N

Write Flow:
1. Write to 2 out of 3 replicas
2. Return success when 2 ACK
3. Third replica updated async

Read Flow:
1. Read from 2 out of 3 replicas
2. Return newest version
3. Read repair if inconsistent

Consistency Levels:
- ONE: R=1, W=1 (eventual consistency)
- QUORUM: R=2, W=2 (balanced)
- ALL: R=3, W=3 (strong consistency)
```

Consistency Models

CAP Theorem

```
Can only guarantee 2 of 3:
- Consistency
- Availability
- Partition Tolerance

┌─────────────────────────────┐
│ Partition Tolerance         │ ← Always needed (network fails)
└─────────────────────────────┘
          │
          │
          │ Choose one:
          │
     ┌────┴────┬─────────┐
     │         │         │
```

```
    ▼         ▼         ▼
Consistency Availability Both
(CP)        (AP)       (CA - impossible in distributed systems)

Examples:
CP: MongoDB, HBase, Redis
AP: Cassandra, DynamoDB, Riak
```

Consistency Levels

1. Strong Consistency

```
Definition: All readers see same value immediately after write

Implementation:
- Synchronous replication
- Read from all replicas
- Write to all replicas

Guarantee: R + W > N and W > N/2

Example: N=3, W=3, R=1
- Must write to all 3 replicas
- Read from any 1 replica sees latest

Use Cases:
- Financial transactions
- Inventory management
- User authentication
```

2. Eventual Consistency

```
Definition: All replicas converge to same value eventually

Implementation:
- Asynchronous replication
- Read from any replica
- Write to any replica

Guarantee: No immediate guarantee

Convergence Time: Typically seconds to minutes

Use Cases:
- Social media feeds
- Product catalogs
```

```
- DNS
- CDN caching
```

3. Read-Your-Writes Consistency

```
Definition: User sees their own writes immediately

Implementation:
- Route reads to same replica as write
- Use session tokens
- Track write version numbers

Example:
1. User writes comment
2. User refreshes page
3. User sees their comment (even if replicas lag)

Use Cases:
- User profiles
- Shopping carts
- Comment systems
```

4. Monotonic Reads

```
Definition: Once you read version V, never see older than V

Implementation:
- Sticky sessions (same replica)
- Version vectors
- Causal consistency

Example:
1. Read version 5
2. Later read version 6 or 5 (never 4)

Use Cases:
- Time-series data
- Audit logs
- Message threads
```

Vector Clocks

```
Track causality across replicas:

Vector Clock: {NodeA: 3, NodeB: 2, NodeC: 1}
```

```
Comparison:
V1 = {A:2, B:1}
V2 = {A:1, B:2}
→ Concurrent (conflict)

V1 = {A:2, B:1}
V2 = {A:3, B:1}
→ V2 happens after V1

Conflict Resolution:
1. Detect concurrent writes
2. Keep both versions
3. Application resolves (last-write-wins, merge, etc.)
```

Failure Detection & Recovery

Types of Failures

```
1. Node Failure:
   - Crash (process dies)
   - Hardware failure
   - OS freeze

2. Network Failure:
   - Partition (split brain)
   - Packet loss
   - High latency

3. Disk Failure:
   - Corruption
   - Out of space
   - I/O errors
```

Failure Detection Mechanisms

1. Heartbeat Protocol

```
Each node sends periodic heartbeats:

Node A → [heartbeat] → Node B (every 1 second)

Node B tracks:
- Last heartbeat time
- Missed heartbeat count

Detection:
```

```
If no heartbeat for 10 seconds:
  → Mark as suspected
If no heartbeat for 30 seconds:
  → Mark as dead
  → Trigger failover
```

2. Phi Accrual Failure Detector

Already covered in Core Components

Recovery Strategies

1. Replica Takeover

```
Scenario: Primary node fails

Before:
Primary (Node A) → Replica (Node B)
                → Replica (Node C)

After Detection:
1. Coordinator detects Node A down
2. Promotes Node B to primary
3. Node B serves reads/writes
4. When Node A recovers:
   - Becomes replica
   - Catches up via anti-entropy

Time to Failover: < 30 seconds
```

2. Hinted Handoff

```
Scenario: Replica temporarily unavailable

Normal:
Client → Coordinator → Replica A ✓
                     → Replica B ✗ (down)
                     → Replica C ✓

With Hinted Handoff:
Client → Coordinator → Replica A ✓
                     → Replica D ✓ (hint for B)
                     → Replica C ✓

When Replica B recovers:
Replica D → Replays hints → Replica B
```

```
Benefits:
- Maintains write availability
- Ensures eventual consistency
- No data loss
```

3. Read Repair

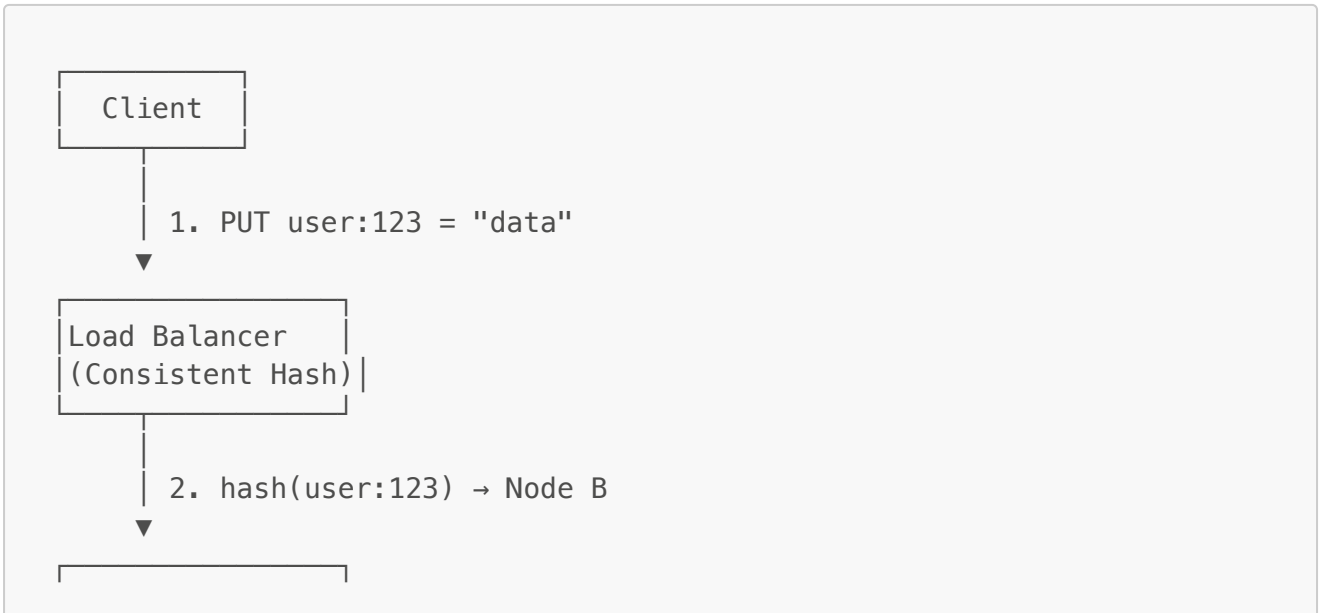
```
Scenario: Inconsistent replicas detected during read

Read Request (R=2):
Client → Coordinator → Replica A (v5)
                     → Replica B (v3)
                     → Replica C (not queried)

Process:
1. Return v5 to client (newest)
2. Background: Update B to v5
3. Background: Check C and update if needed

Benefits:
- Passive repair during reads
- Self-healing
- No extra operations
```

4. Anti-Entropy Repair

Already covered in Core Components (Merkle Trees)

Read/Write Path

Write Path (Detailed)



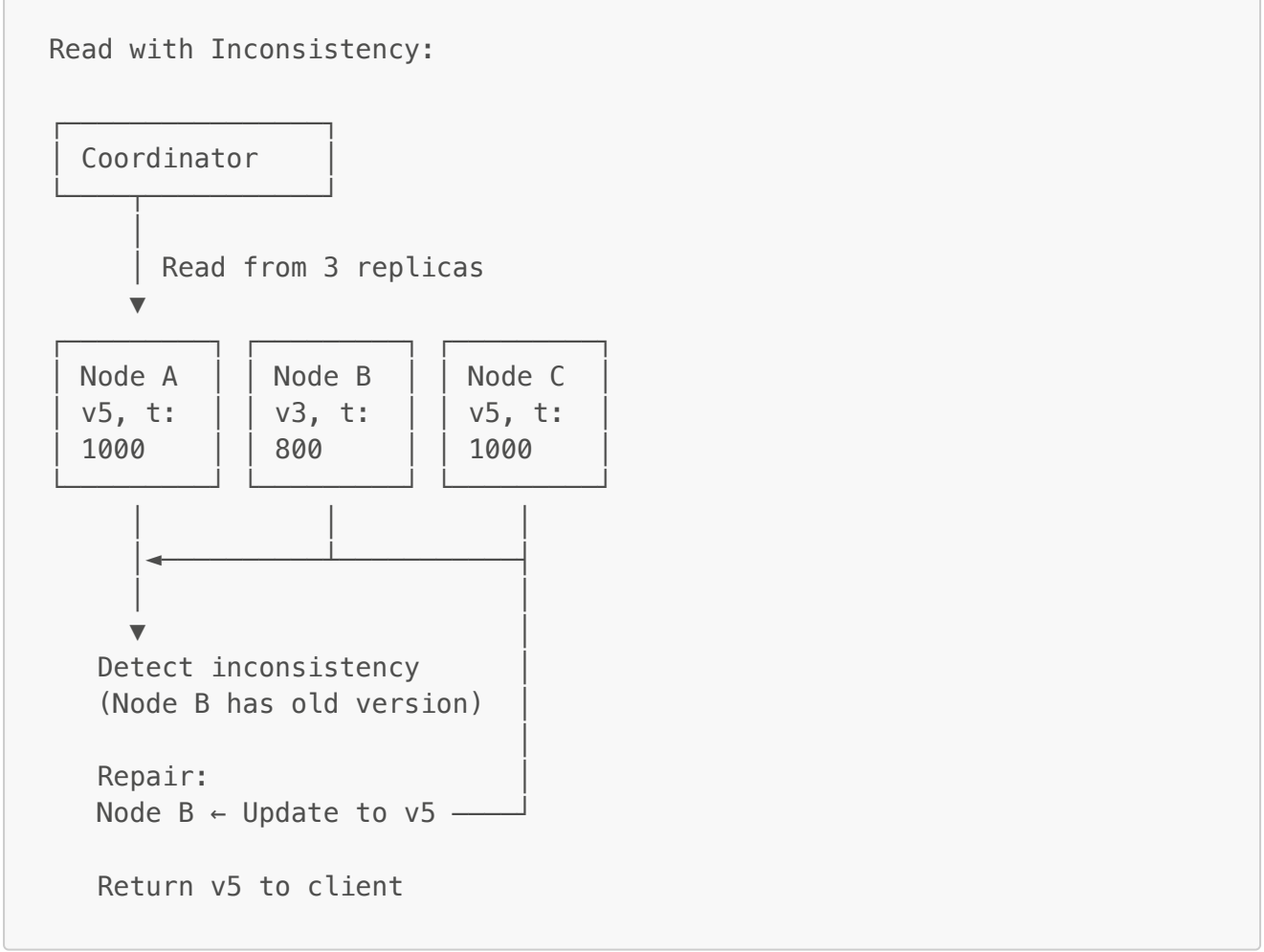
```
┌──────────────────┐
│ Coordinator      │ (Node B)
│ (Node B)         │
└──────────────────┘
      │
      │ 3. Identify replicas: [Node B, Node C, Node D]
      │    Consistency Level: QUORUM (W=2)
      ▼
      ├──────────┬──────────┬──────────┐
      │          │          │          │
      ▼          ▼          ▼          ▼
┌──────────┐ ┌──────────┐ ┌──────────┐
│ Node B   │ │ Node C   │ │ Node D   │
│(Primary) │ │(Replica) │ │(Replica) │
└──────────┘ └──────────┘ └──────────┘
      │            │            │
      │ 4. Write to WAL         │
      │ 5. Write to MemTable    │
      │ 6. Return ACK           │
      │            │            │
      ├◄───────────┤            │
      ├◄─────────────────────────┤
      │                         │
      │ 7. Got 2 ACKs (quorum met)
      ▼
┌──────────────────┐
│  Client          │ ← 8. Success response
└──────────────────┘

Latency Breakdown:
- Network: 1ms (client → coordinator)
- Write WAL: 2ms (disk sync)
- Write MemTable: 0.1ms (in-memory)
- Replication: 3ms (network + remote write)
- Total: ~6-8ms (P99 < 20ms)
```

Read Path (Detailed)

```
┌──────────┐
│  Client  │
└──────────┘
      │
      │ 1. GET user:123
      ▼
┌──────────────────┐
│Load Balancer     │
└──────────────────┘
      │
      │ 2. Route to Node B
      ▼
┌──────────────────┐
```

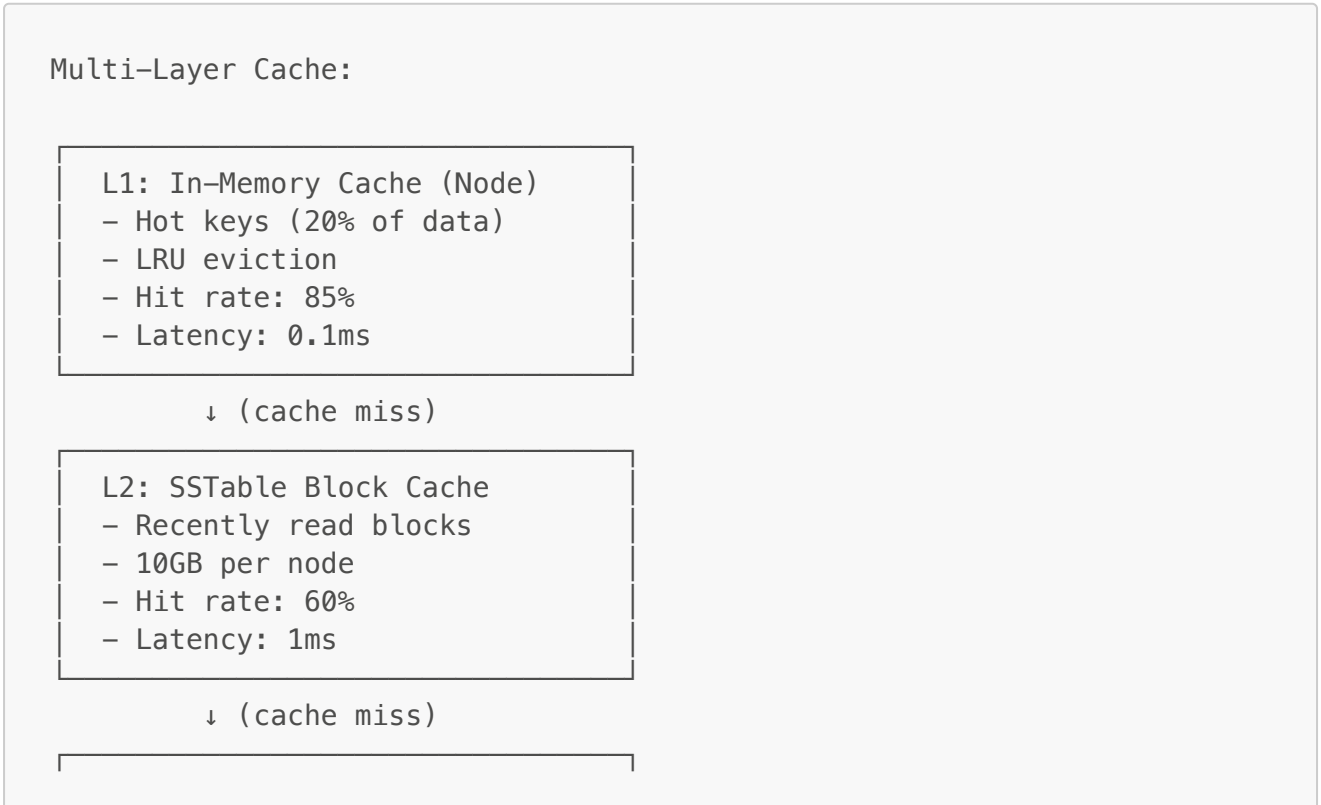
```
┌──────────────────┐
│ Coordinator      │
│ (Node B)         │
└──────────────────┘
        │
        │ 3. Check local cache
        │    Cache: MISS
        │
        │ 4. Consistency Level: QUORUM (R=2)
        │    Send to 2 replicas
        ▼
        ├──────────┬──────────┐
        │          │          │
        ▼          ▼          ▼
┌──────────┐  ┌──────────┐
│ Node B   │  │ Node C   │
└──────────┘  └──────────┘
        │          │
        │ 5. Read path:
        │    a. Check MemTable
        │    b. Check Immutable MemTable
        │    c. Check L0 SSTables (Bloom filter)
        │    d. Check L1 SSTables
        │    e. Found in L1
        │          │
        │◄─────────┤
        │          │
        │ 6. Compare versions
        │    Node B: v5, timestamp: 1000
        │    Node C: v5, timestamp: 1000
        │    (Consistent!)
        │
        ▼
┌──────────────────┐
│  Client          │ ← 7. Return value (v5)
└──────────────────┘

Latency Breakdown:
- Network: 1ms
- Cache check: 0.1ms
- MemTable check: 0.1ms
- SSTable read (L0-L1): 2ms (SSD)
- Total: ~3-5ms (P99 < 10ms)

Optimization: Bloom Filter
- Avoids reading irrelevant SSTables
- ~10 bits per key
- 1% false positive rate
- Saves disk I/O (major win)
```

Read Repair Example



Performance Optimization

1. Caching Strategy



```
│  L3: Disk (SSTables)            │
│  - All data                     │
│  - SSD                          │
│  - Latency: 2-5ms               │
└─────────────────────────────────┘
```

2. Bloom Filters

```
Purpose: Avoid unnecessary disk reads

Structure:
- Bit array (10 bits per key)
- Multiple hash functions (3-5)
- Probabilistic (false positives possible)

Check if key exists:
1. Hash key with H1, H2, H3
2. Check bits at positions
3. If all bits = 1 → Maybe exists
4. If any bit = 0 → Definitely NOT exists

Benefit:
- 99% of non-existent keys filtered
- Saves disk I/O
- Small memory overhead (1.2MB per 1M keys)
```

3. Compaction Strategies

```
Size-Tiered Compaction:
- Merge SSTables of similar size
- Good for write-heavy workloads
- Higher read amplification

Leveled Compaction:
- Each level 10x previous
- No overlap within level
- Better for read-heavy workloads
- Lower space amplification

Time-Window Compaction:
- Merge based on time ranges
- Good for time-series data
- Easy to expire old data
```

4. Batch Operations

```
Benefits:
- Amortize network overhead
- Reduce round trips
- Higher throughput

Example:
Single writes: 100 ops × 5ms = 500ms
Batch write (100): 1 op × 15ms = 15ms

Improvement: 33x faster
```

5. Connection Pooling

```
Problem: Creating connections expensive (10-50ms)

Solution:
┌──────────────────┐
│ Client Pool      │
│ Min: 10 conns    │
│ Max: 100 conns   │
│ Idle timeout:    │
│ 60s              │
└──────────────────┘

Benefits:
- Reuse connections
- Lower latency
- Reduced server load
```

6. Compression

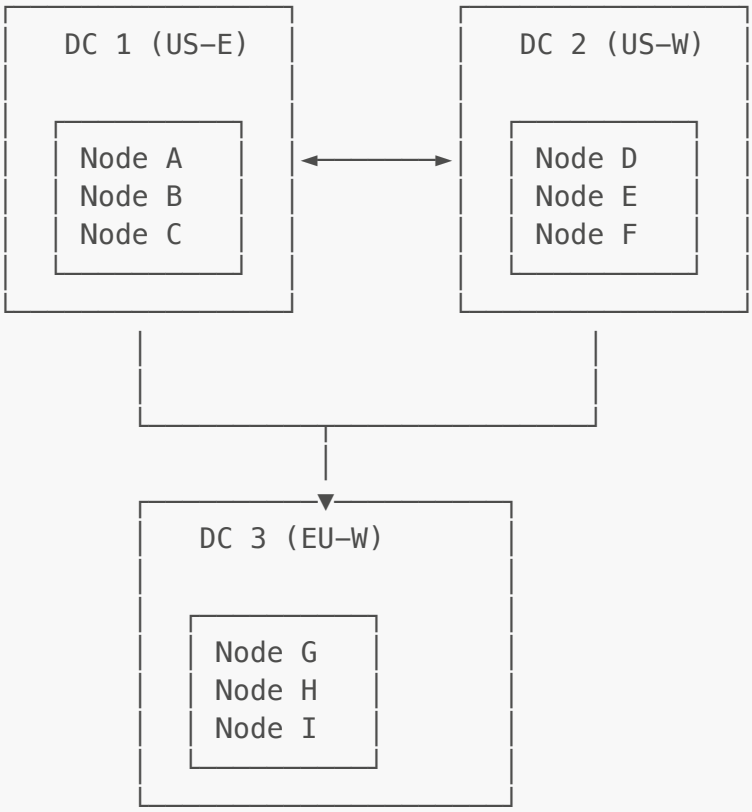
```
Value Compression:
- Snappy (fast, 2-4x compression)
- LZ4 (faster, 2-3x compression)
- Zstandard (balanced, 3-5x compression)

Trade-off:
- CPU overhead vs storage savings
- Latency vs throughput

Recommended: LZ4 for hot data, Zstd for cold data
```

7. Read-Ahead

```
Prefetch adjacent keys:

User requests: user:123
Prefetch: user:124, user:125, user:126

Benefits:
- Faster sequential reads
- Better cache utilization
- Anticipates access patterns

Use Cases:
- Range queries
- Scan operations
- Time-series data
```

Multi-Datacenter Deployment

Architecture




```
Cross-DC Replication:
- Async replication (10-100ms lag)
- Conflict resolution with vector clocks
- Per-DC quorum for availability
```

Replication Strategies

1. Active-Passive

```
Primary DC (Active):
- Handles all writes
- Replicates to passive DCs

Secondary DCs (Passive):
- Read-only
- Failover destination

Pros:
✓ No conflicts
✓ Simpler consistency

Cons:
✗ Write latency for remote users
✗ Underutilized capacity
```

2. Active-Active (Multi-Master)

```
All DCs Accept Writes:
- Users write to nearest DC
- Low latency
- Async replication between DCs

Conflict Resolution:
1. Last-Write-Wins (timestamp)
2. Application-specific merge
3. Vector clocks

Pros:
✓ Low latency globally
✓ Better utilization

Cons:
✗ Conflict resolution complexity
✗ Eventual consistency
```

Geo-Replication

```
Strategy: Place replicas globally

US-East: Primary + 2 replicas
US-West: 2 replicas
```

```
Europe: 2 replicas
Asia: 2 replicas

Read Strategy:
- Route to nearest DC
- Latency: <50ms globally

Write Strategy:
- Write to local DC
- Async replicate globally
- Quorum per DC (local quorum)
```

Network Partition Handling

```
Scenario: DC1 and DC2 partitioned

Option 1: Majority Quorum
- Require majority of DCs to ack
- DC with majority continues
- Minority becomes read-only

Option 2: Per-DC Quorum
- Each DC operates independently
- Merge conflicts on heal
- Higher availability

Option 3: Designated Tie-Breaker
- Third DC decides in split
- Prevents split-brain
- Single point of failure
```

Design Trade-offs

1. Consistency vs Availability

```
Strong Consistency:
Pros:
✓ Simple programming model
✓ No conflicts
✓ Predictable behavior

Cons:
✗ Lower availability (CAP)
✗ Higher latency
✗ Requires synchronous replication

Use When:
```

```
- Financial transactions
- Inventory counts
- User authentication

Eventual Consistency:
Pros:
✓ High availability
✓ Low latency
✓ Geo-replication friendly

Cons:
✗ Complex programming model
✗ Conflict resolution needed
✗ Stale reads possible

Use When:
- Social media
- Product catalogs
- Analytics
```

2. LSM Tree vs B-Tree

```
LSM Tree:
Pros:
✓ Write optimized (sequential I/O)
✓ Better write throughput
✓ Lower write amplification

Cons:
✗ Read amplification (check multiple levels)
✗ Compaction overhead
✗ Space amplification during compaction

Use When:
- Write-heavy workload
- Time-series data
- Logging systems

B-Tree:
Pros:
✓ Read optimized (single lookup)
✓ No compaction needed
✓ Predictable performance

Cons:
✗ Random I/O for writes
✗ Write amplification
✗ Fragmentation

Use When:
```

```
- Read-heavy workload
- Point queries dominant
- Storage efficiency critical
```

3. Replication Factor

```
N=2:
Pros: Lower storage cost, faster writes
Cons: Lower availability, single failure dangerous

N=3 (Recommended):
Pros: Good balance, survive 1 failure
Cons: 3x storage cost

N=5:
Pros: High durability, survive 2 failures
Cons: High cost, slower writes

Decision Factors:
- Data criticality
- Cost constraints
- Availability requirements
```

4. Partition Size

```
Small Partitions (1GB):
Pros:
✓ Fine-grained rebalancing
✓ Faster recovery
✓ Better load distribution

Cons:
✗ More metadata overhead
✗ More coordination
✗ Higher operational complexity

Large Partitions (100GB):
Pros:
✓ Less metadata
✓ Simpler operations
✓ Fewer network hops

Cons:
✗ Coarse rebalancing
✗ Slower recovery
✗ Hot spots
```

Real-World Implementations

1. Amazon DynamoDB

Architecture:

- Multi-master, masterless design
- Consistent hashing for partitioning
- Quorum-based replication (N=3)
- Vector clocks for conflict resolution

Key Features:

- Fully managed service
- Automatic scaling
- Global tables (multi-region)
- Point-in-time recovery

Consistency Model:

- Eventual consistency (default)
- Strong consistency (optional)
- Transactional support (ACID)

Scale:

- Trillions of requests/day
- Millisecond latency at any scale
- Petabytes of data

2. Apache Cassandra

Architecture:

- Ring-based peer-to-peer
- No single point of failure
- Tunable consistency (ONE to ALL)
- Gossip protocol for membership

Key Features:

- Linear scalability
- Multi-datacenter support
- CQL (SQL-like query language)
- Time-series optimized

Consistency Model:

- Eventual consistency
- Tunable per-operation
- Lightweight transactions (Paxos)

Use Cases:

- Apple: 75,000+ nodes
- Netflix: Billions of operations/day
- Instagram: Storing user data

3. Redis

Architecture:

- In-memory data store
- Single-threaded (per core)
- Master-replica replication
- Redis Cluster for sharding

Key Features:

- Sub-millisecond latency
- Rich data structures (lists, sets, sorted sets)
- Pub/Sub messaging
- Lua scripting

Persistence:

- RDB (snapshots)
- AOF (append-only file)
- Hybrid (RDB + AOF)

Use Cases:

- Caching layer
- Session store
- Real-time analytics
- Message queuing

4. etcd

Architecture:

- Raft consensus algorithm
- Strong consistency (CP system)
- Leader-based replication
- Key-value with versioning

Key Features:

- Distributed coordination
- Watch mechanism (notifications)
- Lease/TTL support
- Transaction support

Use Cases:

- Kubernetes: Cluster state
- Service discovery
- Configuration management
- Distributed locking

Summary & Key Takeaways

Core Design Principles

1. **Horizontal Scalability**
    - Use consistent hashing for partitioning
    - Virtual nodes for better distribution
    - Stateless coordinators
2. **High Availability**
    - Replication (N=3 recommended)
    - No single point of failure
    - Automatic failover
3. **Fault Tolerance**
    - Gossip protocol for membership
    - Hinted handoff for temporary failures
    - Anti-entropy for long-term consistency
4. **Performance**
    - LSM tree for write optimization
    - Bloom filters to reduce disk I/O
    - Multi-layer caching
    - Batch operations
5. **Tunability**
    - Configurable consistency levels
    - Flexible replication strategies
    - Adjustable partition sizes

When to Use Key-Value Stores

✓ Good Fit:

- High throughput requirements (100K+ ops/sec)
- Simple data model (key-value pairs)
- Horizontal scalability needed
- High availability critical

- Global distribution required

✕ Poor Fit:

- Complex transactions (use RDBMS)
- Complex queries/joins (use RDBMS)
- Strong consistency required everywhere (use RDBMS)
- Small scale (<1TB, single server sufficient)

Interview Success Tips

1. **Start with Requirements**
    - Scale (TB vs PB)
    - Consistency needs
    - Latency requirements
    - Read/write ratio
2. **Discuss Trade-offs**
    - CAP theorem
    - Consistency vs availability
    - Write vs read optimization
    - Cost vs performance
3. **Draw Clear Diagrams**
    - High-level architecture
    - Data flow
    - Partition distribution
4. **Reference Real Systems**
    - "Similar to DynamoDB's approach..."
    - "Cassandra solves this with..."
    - Shows awareness of production systems
5. **Cover Failure Scenarios**
    - Node failures
    - Network partitions
    - Data center outages
6. **Mention Monitoring**
    - Key metrics to track
    - Alerting strategies
    - Operational concerns

Common Pitfalls to Avoid

❌ Over-engineering for day one
❌ Ignoring CAP theorem
❌ Not considering hot keys
❌ Forgetting about operational complexity
❌ Underestimating network effects
❌ Not planning for data migration
❌ Ignoring monitoring and observability

Further Reading

Essential Papers

- **Dynamo: Amazon's Highly Available Key-value Store** (2007)
- **Cassandra - A Decentralized Structured Storage System** (2010)
- **Bigtable: A Distributed Storage System for Structured Data** (Google, 2006)
- **The Log-Structured Merge-Tree (LSM-Tree)** (1996)

Books

- **Designing Data-Intensive Applications** - Martin Kleppmann
- **Database Internals** - Alex Petrov
- **NoSQL Distilled** - Pramod Sadalage & Martin Fowler

Online Resources

- DynamoDB Paper: aws.amazon.com/dynamodb/
- Cassandra Documentation: cassandra.apache.org
- Redis Documentation: redis.io
- etcd Documentation: etcd.io

Document Version: 1.0 (High-Level Design)
Last Updated: 2025
**Focus