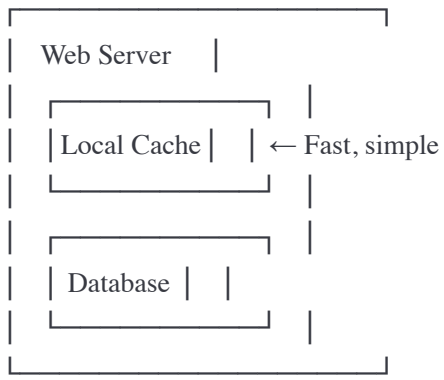
Chapter 19: Distributed Caching and Data Stores

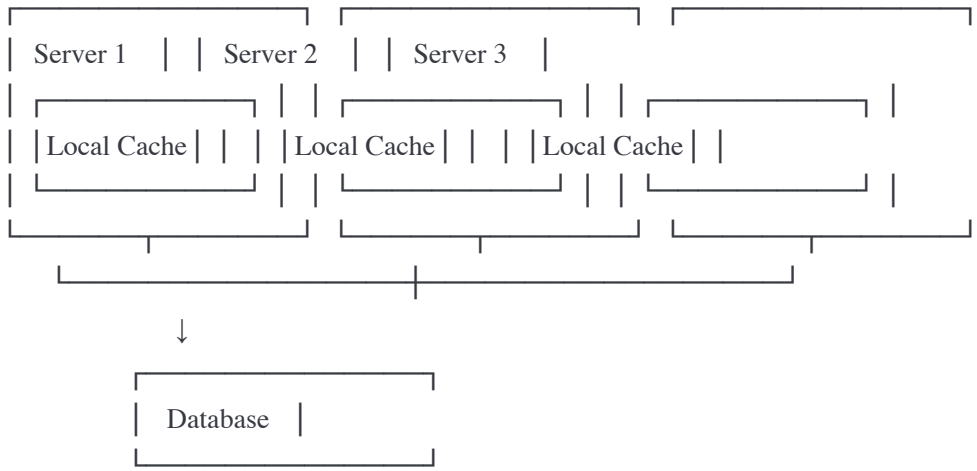
Introduction: The Challenge of Distributed Caching

When you scale horizontally, caching becomes complex.

Single Server (Simple):



Multiple Servers (Complex):



Problems:

1. Cache inconsistency (each server has different data)
2. Cache invalidation (how to update all caches?)
3. Memory waste (same data cached 3 times)
4. Hot spots (one server gets all traffic for popular item)

---

1. Distributed Cache Coherence

The Consistency Problem

Scenario: User updates profile

Time Server 1 Cache Server 2 Cache Database

10:00 name: "John" name: "John" name: "John"
10:01 (User updates via Server 1)
name: "John Doe" name: "John" ✗ name: "John Doe"
Stale data!

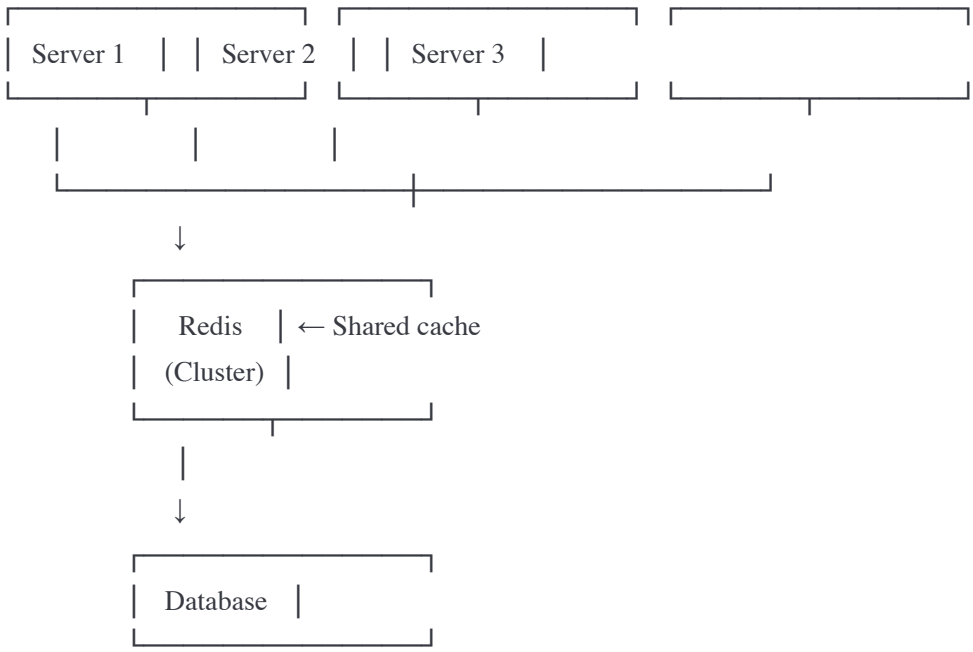
10:02 User visits site, hits Server 2
Sees old name: "John" ❌

This is CACHE COHERENCE problem!

Solution 1: Centralized Cache (Shared Cache)

Use Redis/Memcached as shared cache.

All Servers Share One Cache:



Benefits:
✓ Single source of truth
✓ No inconsistency
✓ Memory efficient
✓ Easy invalidation

Drawbacks:
✗ Network latency (vs local cache)
✗ Redis becomes bottleneck
✗ Single point of failure (mitigated with clustering)

Implementation:

javascript

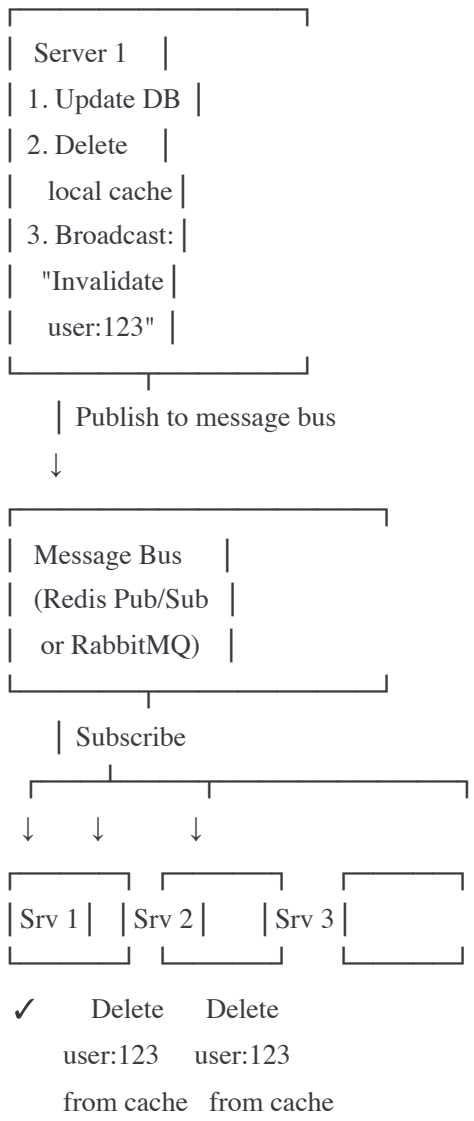
```
const redis = require('redis');
const client = redis.createClient({
  host: 'redis-cluster',
  port: 6379
});

class SharedCacheService {
  constructor(redisClient, database) {
    this.cache = redisClient;
    this.db = database;
  }

  async get(key) {
    // Try cache first
    const cached = await this.cache.get(key);

    if (cached) {
      console.log(`Cache HIT: ${key}`);
      return JSON.parse(cached);
    }

    console.log(`Cache MISS: ${key}`);

    // Fetch from database
    const data = await this.db.get(key);

    if (data) {
      // Store in cache (all servers share this cache)
      await this.cache.setex(key, 3600, JSON.stringify(data));
    }

    return data;
  }

  async set(key, value) {
    // Update database
    await this.db.set(key, value);

    // Update cache (visible to all servers immediately)
    await this.cache.setex(key, 3600, JSON.stringify(value));

    console.log(`Updated cache and DB: ${key}`);
  }

  async invalidate(key) {
    // Delete from cache (affects all servers)
```

```
    await this.cache.del(key);
    console.log(`Invalidated cache: ${key}`);
  }
}

// Usage across multiple servers
// Server 1
const cache1 = new SharedCacheService(client, db);
await cache1.set('user:123', { name: 'John Doe' });

// Server 2 (sees the same cache)
const cache2 = new SharedCacheService(client, db);
const user = await cache2.get('user:123');  // ✓ Gets updated value!
```

Solution 2: Cache Invalidation Messages

Broadcast invalidation to all servers.

User updates profile on Server 1:



All servers now have consistent state!

Implementation:

javascript

```
const redis = require('redis');
const EventEmitter = require('events');

class DistributedCacheWithInvalidation extends EventEmitter {
  constructor(serverId) {
    super();
    this.serverId = serverId;
    this.localCache = new Map();

    // Redis for pub/sub
    this.publisher = redis.createClient();
    this.subscriber = redis.createClient();

    // Subscribe to invalidation messages
    this.subscriber.subscribe('cache-invalidation');

    this.subscriber.on('message', (channel, message) => {
      if (channel === 'cache-invalidation') {
        const { key, sourceServer } = JSON.parse(message);

        // Don't invalidate if we sent the message
        if (sourceServer !== this.serverId) {
          console.log(`[${this.serverId}] Received invalidation for ${key}`);
          this.localCache.delete(key);
        }
      }
    });
  }

  async get(key) {
    // Check local cache
    if (this.localCache.has(key)) {
      console.log(`[${this.serverId}] Local cache HIT: ${key}`);
      return this.localCache.get(key);
    }

    console.log(`[${this.serverId}] Local cache MISS: ${key}`);

    // Fetch from database
    const data = await database.get(key);

    if (data) {
      this.localCache.set(key, data);
    }

    return data;
```

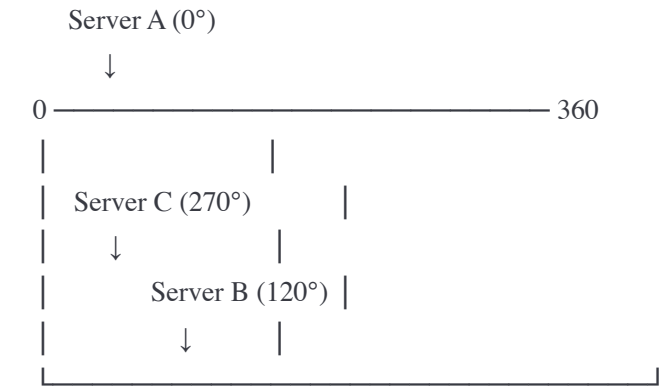
```
  }

  async set(key, value) {
    // Update database
    await database.set(key, value);

    // Update local cache
    this.localCache.set(key, value);

    // Broadcast invalidation to other servers
    await this.publisher.publish('cache-invalidation', JSON.stringify({
      key: key,
      sourceServer: this.serverId,
      timestamp: Date.now()
    }));

    console.log(`[${this.serverId}] Updated and broadcasted invalidation`);
  }

  async invalidate(key) {
    // Delete from local cache
    this.localCache.delete(key);

    // Broadcast to other servers
    await this.publisher.publish('cache-invalidation', JSON.stringify({
      key: key,
      sourceServer: this.serverId,
      timestamp: Date.now()
    }));
  }
}

// Usage on different servers
const server1Cache = new DistributedCacheWithInvalidation('server-1');
const server2Cache = new DistributedCacheWithInvalidation('server-2');
const server3Cache = new DistributedCacheWithInvalidation('server-3');

// Server 1 updates user
await server1Cache.set('user:123', { name: 'John Doe' });
// → Broadcasts invalidation
// → Server 2 and 3 delete from their local caches

// Server 2 reads user (after invalidation)
const user = await server2Cache.get('user:123');
// → Local cache empty (invalidated)
```

```
// → Fetches from database
// → Gets updated value ✓
```

Solution 3: Consistent Hashing (Distribute Keys)

Each key always goes to same cache server.

Consistent Hashing Ring:



Hash key to find server:
- hash("user:123") = 45° → Server B
- hash("user:456") = 200° → Server C
- hash("user:789") = 310° → Server A

Same key always goes to same server!
No duplication, no inconsistency

Benefits:
✓ No duplicate data
✓ Consistent location
✓ Scales horizontally

Drawbacks:
✗ Uneven distribution (hotspots)
✗ Adding server moves data

Implementation (from Chapter 7):

javascript

```
const crypto = require('crypto');

class ConsistentHashCache {
  constructor(servers) {
    this.servers = new Map();
    this.ring = [];
    this.virtualNodes = 150;  // Per server

    servers.forEach(server => this.addServer(server));
  }

  hash(key) {
    return parseInt(
      crypto.createHash('md5').update(key).digest('hex').substring(0, 8),
      16
    );
  }

  addServer(server) {
    // Add virtual nodes for better distribution
    for (let i = 0; i < this.virtualNodes; i++) {
      const hash = this.hash(`${server.id}:${i}`);
      this.ring.push({ hash, server });
    }

    // Sort ring by hash
    this.ring.sort((a, b) => a.hash - b.hash);

    this.servers.set(server.id, server);
    console.log(`Added server ${server.id} to ring`);
  }

  getServer(key) {
    const keyHash = this.hash(key);

    // Find first server clockwise from key
    for (let i = 0; i < this.ring.length; i++) {
      if (this.ring[i].hash >= keyHash) {
        return this.ring[i].server;
      }
    }

    // Wrap around
    return this.ring[0].server;
  }
```

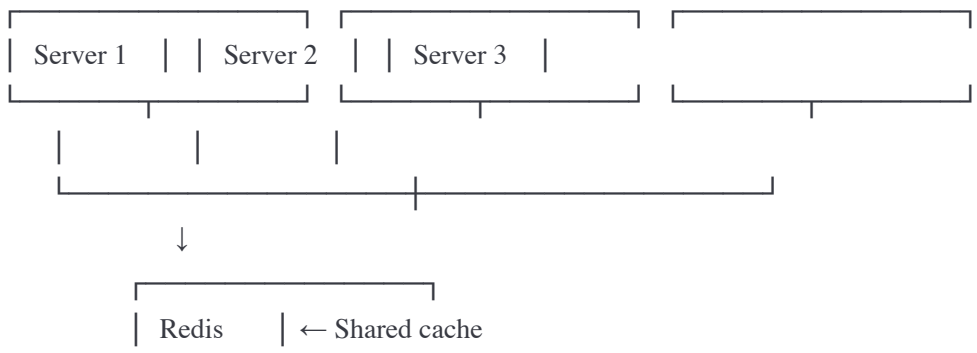
```
  async get(key) {
    const server = this.getServer(key);
    console.log(`Key ${key} → Server ${server.id}`);

    // Get from specific server
    return await server.cache.get(key);
  }

  async set(key, value) {
    const server = this.getServer(key);
    console.log(`Key ${key} → Server ${server.id}`);

    // Set on specific server
    await server.cache.set(key, value);
  }
}

// Create cache cluster
const cacheCluster = new ConsistentHashCache([
  { id: 'cache-1', cache: redisClient1 },
  { id: 'cache-2', cache: redisClient2 },
  { id: 'cache-3', cache: redisClient3 }
]);

// All requests for same key go to same server
await cacheCluster.set('user:123', userData);
await cacheCluster.get('user:123');  // Always cache-1 (example)

// No cache inconsistency!
// No duplicate storage!
```

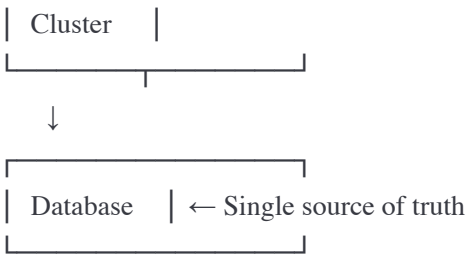
---

2. Write-Through and Write-Behind at Scale

Write-Through Caching (Distributed)

Architecture:





Write Flow:

1. Write to cache
2. Write to database (synchronous)
3. Both must succeed

Implementation:

javascript

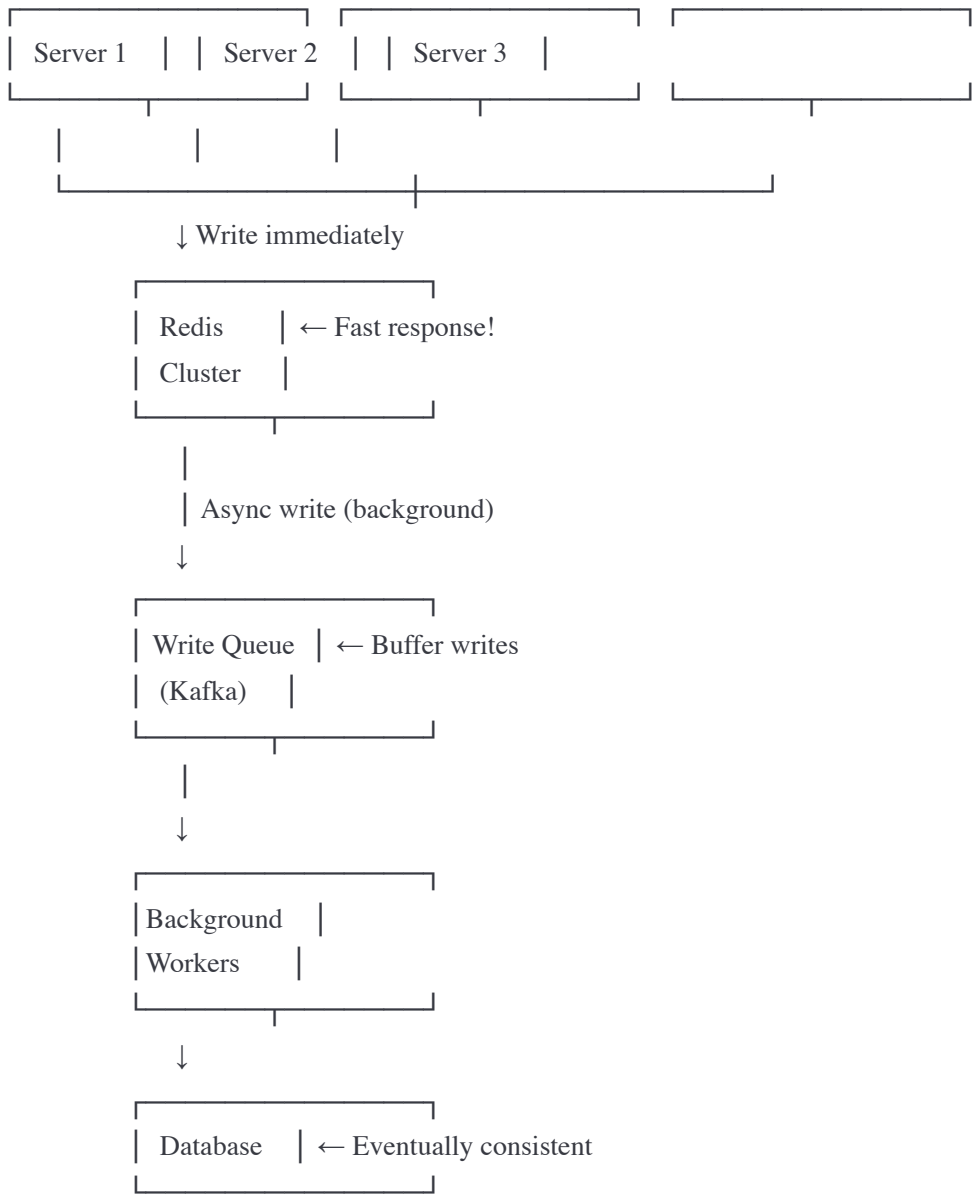
```
class DistributedWriteThroughCache {
  constructor(redisCluster, database) {
    this.cache = redisCluster;
    this.db = database;
  }

  async get(key) {
    try {
      // Try cache first
      const cached = await this.cache.get(key);

      if (cached) {
        console.log(`Cache HIT: ${key}`);
        return JSON.parse(cached);
      }

      console.log(`Cache MISS: ${key}`);

      // Fetch from database
      const data = await this.db.query(
        'SELECT * FROM users WHERE id = ?',
        [key]
      );

      if (data) {
        // Populate cache
        await this.cache.setex(key, 3600, JSON.stringify(data));
      }

      return data;

    } catch (error) {
      console.error('Cache error, falling back to database:', error);
      // Degrade gracefully: bypass cache
      return await this.db.query('SELECT * FROM users WHERE id = ?', [key]);
    }
  }

  async set(key, value) {
    try {
      // Write-through: Update BOTH cache and database
      const dbPromise = this.db.query(
        'UPDATE users SET data = ? WHERE id = ?',
        [JSON.stringify(value), key]
      );
```

```
      const cachePromise = this.cache.setex(
        key,
        3600,
        JSON.stringify(value)
      );

      // Wait for both (atomic)
      await Promise.all([dbPromise, cachePromise]);

      console.log(`Write-through complete: ${key}`);

    } catch (error) {
      console.error('Write-through failed:', error);

      // Ensure consistency: invalidate cache on error
      await this.cache.del(key);

      throw error;
    }
  }
}

// Usage across multiple servers
const cache = new DistributedWriteThroughCache(redisCluster, database);

// Server 1 writes
await cache.set('user:123', { name: 'John Doe', email: 'john@example.com' });

// Server 2 reads (gets updated value from shared cache)
const user = await cache.get('user:123');
console.log(user);  // ✓{ name: 'John Doe', ... }
```

Write-Behind Caching (Distributed)

Asynchronous database writes for better performance.

Architecture with Write-Behind:



Write Flow:

1. Write to cache (fast!)
2. Return success to user
3. Queue database write
4. Background worker processes queue

Implementation:

javascript

```
const { Kafka } = require('kafkajs');

class DistributedWriteBehindCache {
  constructor(redisCluster, writeQueue) {
    this.cache = redisCluster;
    this.queue = writeQueue;
  }

  async get(key) {
    // Same as write-through
    const cached = await this.cache.get(key);

    if (cached) {
      return JSON.parse(cached);
    }

    const data = await database.get(key);

    if (data) {
      await this.cache.setex(key, 3600, JSON.stringify(data));
    }

    return data;
  }

  async set(key, value) {
    // Write to cache immediately (fast!)
    await this.cache.setex(key, 3600, JSON.stringify(value));

    console.log(`Write-behind: Cached ${key} (0.5ms)`);

    // Queue database write (async, don't wait)
    await this.queue.send({
      topic: 'cache-writes',
      messages: [{
        key: key,
        value: JSON.stringify({
          operation: 'set',
          key: key,
          value: value,
          timestamp: Date.now()
        })
      }]
    });

    console.log(`Write-behind: Queued DB write for ${key}`);
```

```
    // Return immediately (user doesn't wait for DB!)
    return { success: true };
  }
}

// Background Worker (processes queue)
class WriteBehindWorker {
  constructor(database) {
    this.db = database;
    this.kafka = new Kafka({ brokers: ['kafka:9092'] });
    this.batchSize = 100;
    this.batchTimeout = 5000;  // 5 seconds
  }

  async start() {
    const consumer = this.kafka.consumer({ groupId: 'cache-write-workers' });
    await consumer.connect();
    await consumer.subscribe({ topic: 'cache-writes' });

    let batch = [];
    let lastFlush = Date.now();

    await consumer.run({
      eachMessage: async ({ message }) => {
        const write = JSON.parse(message.value);
        batch.push(write);

        // Flush if batch full or timeout
        if (batch.length >= this.batchSize ||
            Date.now() - lastFlush > this.batchTimeout) {
          await this.flushBatch(batch);
          batch = [];
          lastFlush = Date.now();
        }
      }
    });
  }

  async flushBatch(batch) {
    if (batch.length === 0) return;

    console.log(`Flushing batch of ${batch.length} writes to database...`);

    try {
      // Begin transaction
      await this.db.query('BEGIN');
```

```
    // Execute all writes in batch
    for (const write of batch) {
      await this.db.query(
        'UPDATE users SET data = ? WHERE id = ?',
        [JSON.stringify(write.value), write.key]
      );
    }

    // Commit
    await this.db.query('COMMIT');

    console.log(`✓ Batch committed (${batch.length} writes)`);

  } catch (error) {
    console.error('Batch write failed:', error);
    await this.db.query('ROLLBACK');

    // Re-queue failed writes
    // In production: send to DLQ
  }
 }
}

// Start worker
const worker = new WriteBehindWorker(database);
worker.start();

// Usage
const cache = new DistributedWriteBehindCache(redisCluster, kafka.producer());

// Fast writes!
await cache.set('user:123', { name: 'John' });  // Returns in 1ms
await cache.set('user:456', { name: 'Jane' });  // Returns in 1ms

// Database updated in background (batched)
// User doesn't wait!
```

Write-Behind Challenges

Challenges:

1. DATA LOSS RISK
   Cache crashes before DB write
   → Data lost!

Solution: Persistent cache (Redis with AOF)
+ Message queue durability

2. WRITE ORDERING
Updates must be applied in order

Solution: Partition queue by key
Same key → same partition → ordered

3. READ-YOUR-WRITES
User writes, immediately reads
Sees old data from database!

Solution: Read from cache after write

4. CONSISTENCY
Cache and DB temporarily inconsistent

Solution: Acceptable for non-critical data only

Read-Your-Writes Implementation:

javascript

```
class WriteBehindWithConsistency {
  constructor(cache, queue, db) {
    this.cache = cache;
    this.queue = queue;
    this.db = db;
    this.recentWrites = new Map();  // Track recent writes per session
  }

  async set(key, value, sessionId) {
    // Write to cache
    await this.cache.setex(key, 3600, JSON.stringify(value));

    // Track write for this session
    if (!this.recentWrites.has(sessionId)) {
      this.recentWrites.set(sessionId, new Set());
    }
    this.recentWrites.get(sessionId).add(key);

    // Queue DB write
    await this.queue.send({ key, value });

    // Cleanup after 60 seconds
    setTimeout(() => {
      this.recentWrites.get(sessionId)?.delete(key);
    }, 60000);
  }

  async get(key, sessionId) {
    // Check if this session recently wrote this key
    if (this.recentWrites.get(sessionId)?.has(key)) {
      // Read from cache (has latest value)
      console.log('Read-your-writes: Using cache');
      const cached = await this.cache.get(key);
      return JSON.parse(cached);
    }

    // Normal flow: try cache, then DB
    const cached = await this.cache.get(key);
    if (cached) {
      return JSON.parse(cached);
    }

    return await this.db.get(key);
  }
}
```

```
// Usage
const cache = new WriteBehindWithConsistency(redis, kafka, db);

// User writes
await cache.set('user:123', { name: 'Updated' }, 'session-abc');

// Immediately reads (same session)
const user = await cache.get('user:123', 'session-abc');
// → Reads from cache (guaranteed to see own write) ✓

// Different user reads
const user2 = await cache.get('user:123', 'session-xyz');
// → May read from DB (might not see update yet)
// → Eventually consistent
```

---

3. Session Management in Distributed Systems

The Session Problem

Problem: HTTP is stateless

User logs in on Server 1:

```
┌──────────────────┐
│ Server 1         │
│ Session: {       │
│   userId: 123    │
│   loggedIn: ✓    │
│ }                │
└──────────────────┘
```

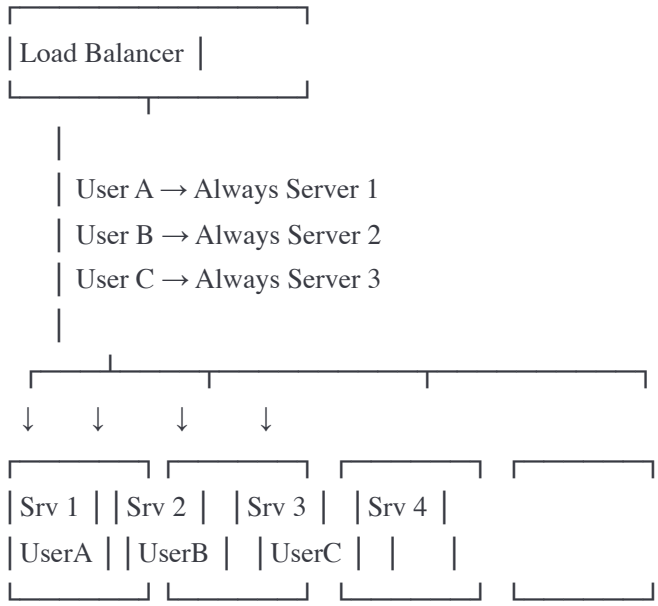
Next request hits Server 2:

```
┌──────────────────┐
│ Server 2         │
│ No session!      │ ← Doesn't know user is logged in
│ Redirects to     │
│ login page ✗     │
└──────────────────┘
```

User thinks: "I just logged in! Why ask again?"

Solution 1: Sticky Sessions (Session Affinity)

Load Balancer pins user to same server:



Pros:

✓ Simple

✓ Local session (fast)

Cons:

✗ Uneven load (if User A very active)

✗ If Server 1 crashes, User A loses session

✗ Can't scale horizontally easily

Implementation:

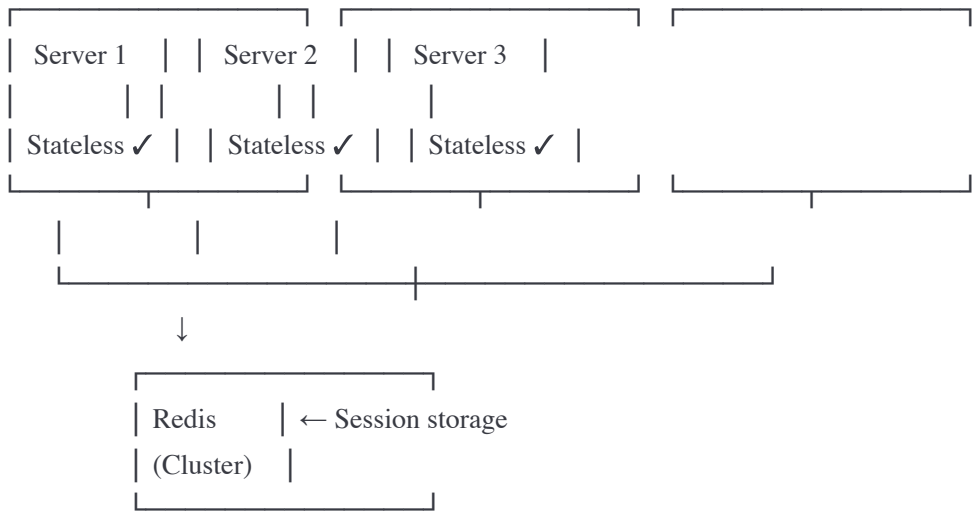
javascript

```
// Load balancer with sticky sessions
class StickyLoadBalancer {
  constructor(servers) {
    this.servers = servers;
    this.userServerMap = new Map();
  }

  getServer(userId) {
    // Check if user already mapped
    if (this.userServerMap.has(userId)) {
      const server = this.userServerMap.get(userId);

      // Check if server still healthy
      if (server.healthy) {
        return server;
      }

      // Server unhealthy, remap user
      console.log(`Server ${server.id} unhealthy, remapping user ${userId}`);
      this.userServerMap.delete(userId);
    }

    // Map to least loaded healthy server
    const healthyServers = this.servers.filter(s => s.healthy);
    const server = healthyServers.reduce((min, s) =>
      s.activeUsers < min.activeUsers ? s : min
    );

    this.userServerMap.set(userId, server);
    server.activeUsers++;

    console.log(`Mapped user ${userId} to server ${server.id}`);
    return server;
  }

  route(request) {
    const userId = request.userId;
    const server = this.getServer(userId);

    // Route to specific server
    return server.handle(request);
  }
}
```

Solution 2: Shared Session Store (Recommended)

Store sessions in Redis (centralized).

All Servers Share Session Store:



Benefits:

✓ Any server can handle any request

✓ True horizontal scaling

✓ Server crash doesn't lose sessions

✓ Easy to add/remove servers

Drawbacks:

✗ Network call for every session access

✗ Redis is critical dependency

Implementation:

javascript

```
const session = require('express-session');
const RedisStore = require('connect-redis')(session);
const redis = require('redis');

const redisClient = redis.createClient({
  host: 'redis-cluster',
  port: 6379
});

// Express session middleware
app.use(session({
  store: new RedisStore({ client: redisClient }),
  secret: 'your-secret-key',
  resave: false,
  saveUninitialized: false,
  cookie: {
    maxAge: 24 * 60 * 60 * 1000,  // 24 hours
    secure: true,
    httpOnly: true
  }
}));

// Login endpoint
app.post('/login', async (req, res) => {
  const { username, password } = req.body;

  const user = await authenticate(username, password);

  if (user) {
    // Store in Redis session (shared across all servers)
    req.session.userId = user.id;
    req.session.username = user.username;
    req.session.role = user.role;

    console.log('Session created in Redis:', req.session.id);

    res.json({ success: true });
  } else {
    res.status(401).json({ error: 'Invalid credentials' });
  }
});

// Protected endpoint
app.get('/dashboard', (req, res) => {
  if (!req.session.userId) {
    return res.status(401).json({ error: 'Not logged in' });
```

```
  }

  // Session available on ANY server!
  res.json({
    userId: req.session.userId,
    username: req.session.username
  });
});

// Logout
app.post('/logout', (req, res) => {
  req.session.destroy((err) => {
    if (err) {
      console.error('Failed to destroy session:', err);
    }
    res.json({ success: true });
  });
});
```

Solution 3: JWT (Stateless Sessions)

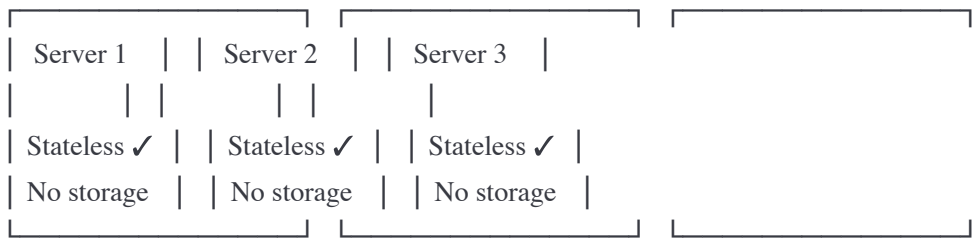
No server-side storage needed!

JWT Flow:

1. Login:
   User → Server
   Server validates
   Server creates JWT: {userId: 123, exp: ...}
   Server signs: JWT + secret = signed token
   Server → User (token)

2. Subsequent Requests:
   User → Server (token in header)
   Server verifies signature
   Server decodes: {userId: 123}
   Server processes request

No session storage needed!
Token contains all info!



No Redis needed!
Any server can validate any token!

Implementation:

javascript

```
const jwt = require('jsonwebtoken');

const SECRET_KEY = 'your-secret-key';

// Login (create JWT)
app.post('/login', async (req, res) => {
  const { username, password } = req.body;

  const user = await authenticate(username, password);

  if (user) {
    // Create JWT token
    const token = jwt.sign(
      {
        userId: user.id,
        username: user.username,
        role: user.role
      },
      SECRET_KEY,
      {
        expiresIn: '24h',
        issuer: 'my-app'
      }
    );

    res.json({ token });
  } else {
    res.status(401).json({ error: 'Invalid credentials' });
  }
});

// Authentication middleware
function authenticateJWT(req, res, next) {
  const token = req.headers.authorization?.replace('Bearer ', '');

  if (!token) {
    return res.status(401).json({ error: 'No token provided' });
  }

  try {
    // Verify and decode token
    const decoded = jwt.verify(token, SECRET_KEY);

    req.user = decoded;
    next();
```

```
  } catch (error) {
    res.status(401).json({ error: 'Invalid token' });
  }
}

// Protected endpoint
app.get('/dashboard', authenticateJWT, (req, res) => {
  // User info from JWT (no database/cache lookup needed!)
  res.json({
    userId: req.user.userId,
    username: req.user.username,
    role: req.user.role
  });
});

// No session storage, completely stateless!
```

Session Management Comparison

Method	Storage	Scaling	Security
Sticky Sessions (Server memory)	Local	Poor	Low
Shared Store (Redis)	Redis	Excellent	Good
JWT (Stateless)	None	Excellent	Medium

Recommendation:
- Small apps: Sticky sessions (simple)
- Medium apps: Redis sessions (scalable)
- APIs: JWT (stateless, microservices-friendly)
- Hybrid: Redis + JWT (best of both)

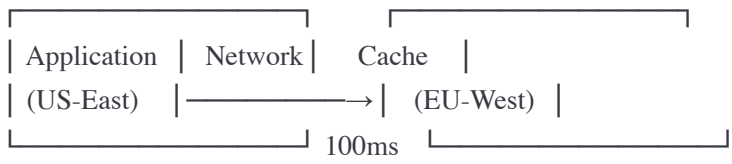
---

4. Data Locality and Hotspot Management

Data Locality

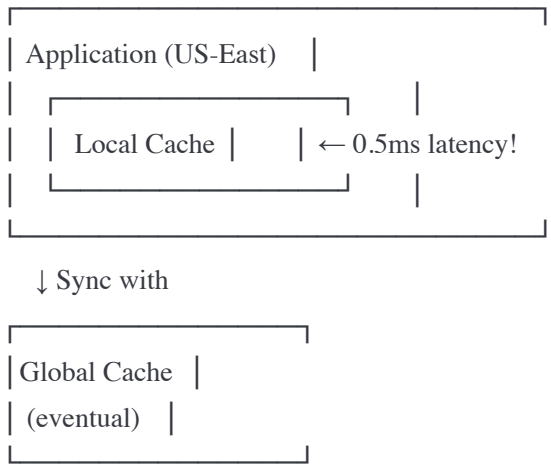
Concept: Keep data close to computation.

Poor Locality:



Every cache access: 100ms latency!

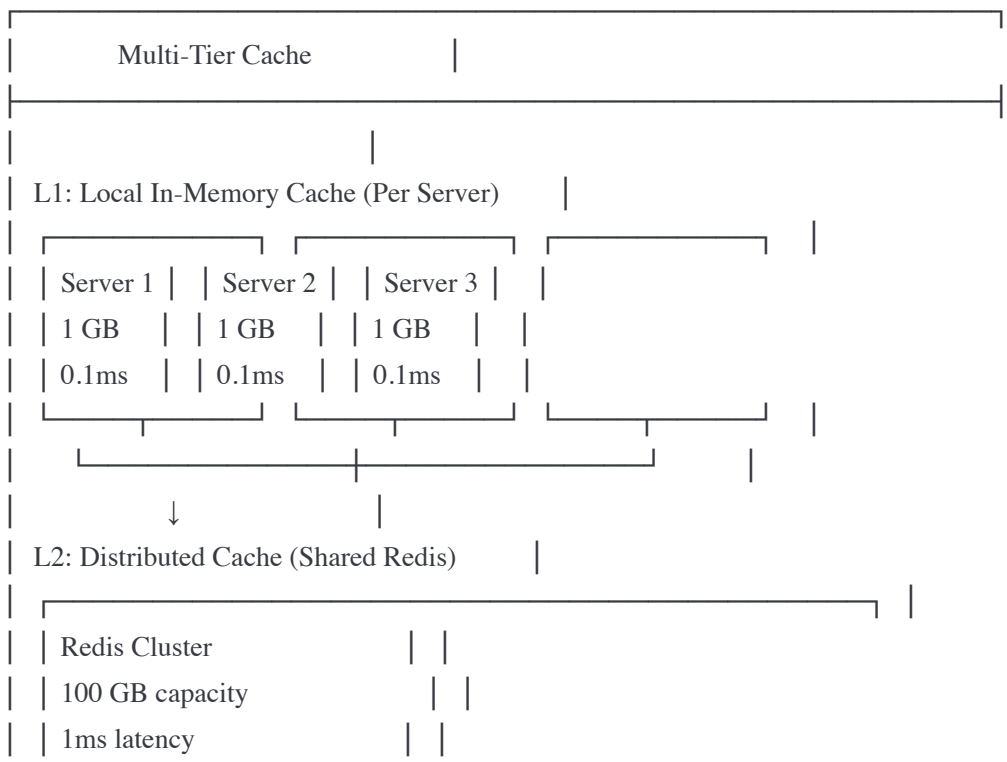
Good Locality:

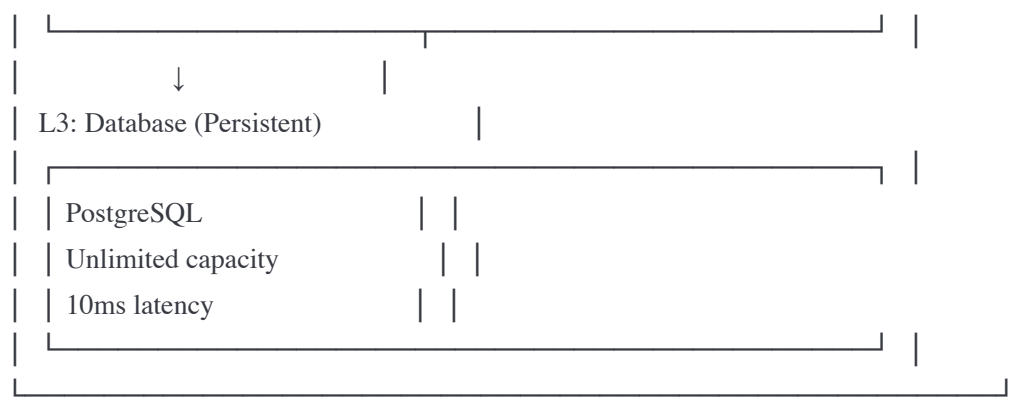


100x faster!

Multi-Tier Caching

Caching Hierarchy (Bring Data Closer):





Lookup Flow:
1. Check L1 (local) → 90% hit rate, 0.1ms
2. Check L2 (Redis) → 9% hit rate, 1ms
3. Check L3 (DB) → 1% hit rate, 10ms

Average latency: $(0.9 \times 0.1) + (0.09 \times 1) + (0.01 \times 10)$
$= 0.09 + 0.09 + 0.1 = 0.28$ms

vs Redis only: 1ms
vs Database only: 10ms

35x faster than Redis, 35x faster than DB!

Implementation:

javascript

```
const NodeCache = require('node-cache');
const redis = require('redis');

class MultiTierCache {
  constructor() {
    // L1: Local in-memory cache (per server instance)
    this.l1Cache = new NodeCache({
      stdTTL: 300,  // 5 minutes
      checkperiod: 60,
      maxKeys: 10000  // Limit memory usage
    });

    // L2: Shared Redis cluster
    this.l2Cache = redis.createClient({
      host: 'redis-cluster',
      port: 6379
    });

    // Statistics
    this.stats = {
      l1Hits: 0,
      l2Hits: 0,
      dbHits: 0,
      totalRequests: 0
    };
  }

  async get(key) {
    this.stats.totalRequests++;

    // L1: Check local cache
    const l1Value = this.l1Cache.get(key);
    if (l1Value !== undefined) {
      this.stats.l1Hits++;
      console.log(`L1 HIT: ${key} (0.1ms)`);
      return l1Value;
    }

    // L2: Check Redis
    const l2Value = await this.l2Cache.get(key);
    if (l2Value) {
      this.stats.l2Hits++;
      console.log(`L2 HIT: ${key} (1ms)`);

      const parsed = JSON.parse(l2Value);
```

```
    // Promote to L1
    this.l1Cache.set(key, parsed);

    return parsed;
  }

  // L3: Database
  this.stats.dbHits++;
  console.log(`DB HIT: ${key} (10ms)`);

  const dbValue = await database.get(key);

  if (dbValue) {
    // Populate both caches
    this.l1Cache.set(key, dbValue);
    await this.l2Cache.setex(key, 3600, JSON.stringify(dbValue));
  }

  return dbValue;
}

async set(key, value) {
  // Write to database
  await database.set(key, value);

  // Update L2 cache
  await this.l2Cache.setex(key, 3600, JSON.stringify(value));

  // Update L1 cache
  this.l1Cache.set(key, value);

  console.log(`Updated all cache tiers: ${key}`);
}

async invalidate(key) {
  // Invalidate all tiers
  this.l1Cache.del(key);
  await this.l2Cache.del(key);

  console.log(`Invalidated: ${key}`);
}

getStats() {
  const total = this.stats.totalRequests;

  return {
    totalRequests: total,
```

```
      l1HitRate: ((this.stats.l1Hits / total) * 100).toFixed(2) + '%',
      l2HitRate: ((this.stats.l2Hits / total) * 100).toFixed(2) + '%',
      dbHitRate: ((this.stats.dbHits / total) * 100).toFixed(2) + '%',
      avgLatency: (
        (this.stats.l1Hits * 0.1) +
        (this.stats.l2Hits * 1) +
        (this.stats.dbHits * 10)
      ) / total
    };
  }
}

// Usage
const cache = new MultiTierCache();

// Simulate traffic
for (let i = 0; i < 1000; i++) {
  // 80% requests for popular items (cache hits)
  const userId = Math.random() < 0.8
    ? Math.floor(Math.random() * 10)  // Popular users
    : Math.floor(Math.random() * 1000);  // Random users

  await cache.get(`user:${userId}`);
}

console.log('Cache Statistics:', cache.getStats());

// Output:
// Cache Statistics: {
//   totalRequests: 1000,
//   l1HitRate: '89.2%',  (most requests)
//   l2HitRate: '8.5%',   (some misses)
//   dbHitRate: '2.3%',   (rare)
//   avgLatency: 0.32ms   (very fast!)
// }
```

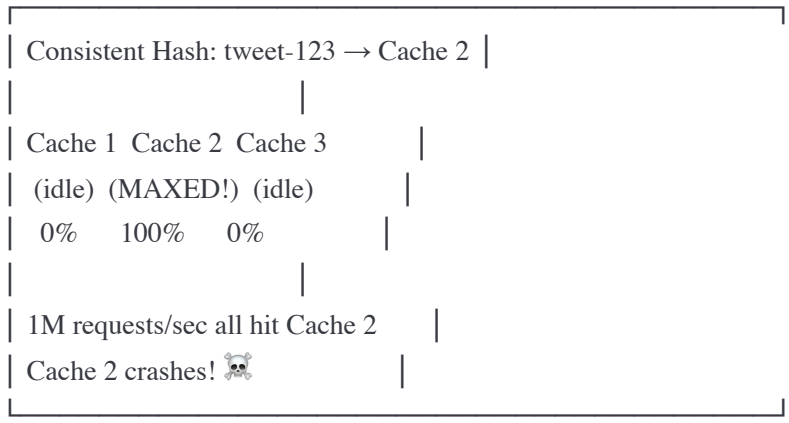
Hotspot Management

Problem: Popular keys overwhelm single cache server.

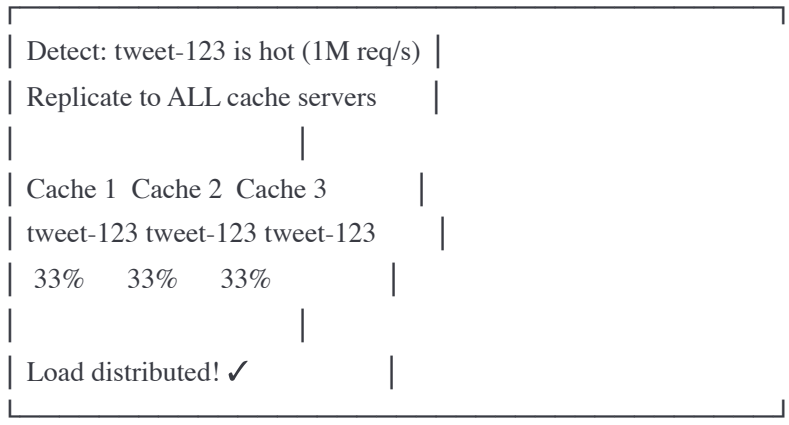
Problem: Celebrity Tweet

Celebrity tweets (1M followers):
Tweet ID: tweet-123

Without hotspot handling:



Solution: Hotspot Replication



Hotspot Detection and Mitigation

javascript

```
class HotspotDetector {
  constructor(threshold = 1000) {
    this.accessCounts = new Map();
    this.hotspotThreshold = threshold;  // requests/sec
    this.hotKeys = new Set();
    this.window = 1000;  // 1 second window
  }

  recordAccess(key) {
    const now = Date.now();

    if (!this.accessCounts.has(key)) {
      this.accessCounts.set(key, []);
    }

    const timestamps = this.accessCounts.get(key);
    timestamps.push(now);

    // Remove old timestamps (outside window)
    const cutoff = now - this.window;
    const filtered = timestamps.filter(ts => ts > cutoff);
    this.accessCounts.set(key, filtered);

    // Check if hotspot
    if (filtered.length > this.hotspotThreshold) {
      if (!this.hotKeys.has(key)) {
        console.log(`🔥 HOTSPOT DETECTED: ${key} (${filtered.length} req/s)`);
        this.hotKeys.add(key);
        this.emit('hotspot', key);
      }
    } else if (filtered.length < this.hotspotThreshold * 0.5) {
      // Cool down
      if (this.hotKeys.has(key)) {
        console.log(`❄️ Hotspot cooled: ${key}`);
        this.hotKeys.delete(key);
        this.emit('cooldown', key);
      }
    }
  }

  isHotspot(key) {
    return this.hotKeys.has(key);
  }
}

class CacheWithHotspotHandling {
```

```
constructor(cacheServers) {
  this.cacheServers = cacheServers;
  this.hotspotDetector = new HotspotDetector(1000);
  this.replicatedKeys = new Set();

  // Listen for hotspot events
  this.hotspotDetector.on('hotspot', (key) => {
    this.replicateKey(key);
  });

  this.hotspotDetector.on('cooldown', (key) => {
    this.unreplicateKey(key);
  });
}

async get(key) {
  // Record access for hotspot detection
  this.hotspotDetector.recordAccess(key);

  if (this.replicatedKeys.has(key)) {
    // Hotspot: Load balance across all servers
    const server = this.cacheServers[
      Math.floor(Math.random() * this.cacheServers.length)
    ];

    console.log(`Hotspot ${key} → Random server ${server.id}`);
    return await server.get(key);

  } else {
    // Normal: Consistent hashing
    const server = this.getServerForKey(key);
    console.log(`Normal ${key} → Server ${server.id}`);
    return await server.get(key);
  }
}

async replicateKey(key) {
  console.log(`Replicating hotspot key: ${key}`);

  // Get value from primary server
  const primaryServer = this.getServerForKey(key);
  const value = await primaryServer.get(key);

  // Replicate to all servers
  for (const server of this.cacheServers) {
    if (server !== primaryServer) {
      await server.set(key, value);
```

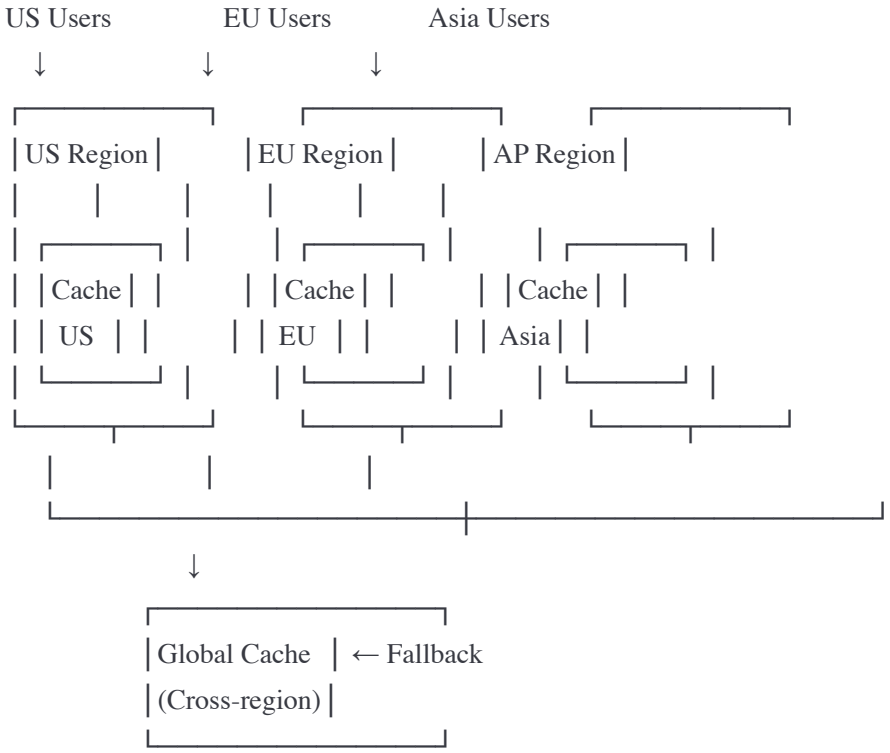
```
      }
    }

    this.replicatedKeys.add(key);
    console.log(`✓ Key ${key} replicated to all ${this.cacheServers.length} servers`);
  }

  async unreplicateKey(key) {
    console.log(`Removing replication for cooled key: ${key}`);

    // Remove from all servers except primary
    const primaryServer = this.getServerForKey(key);

    for (const server of this.cacheServers) {
      if (server !== primaryServer) {
        await server.delete(key);
      }
    }

    this.replicatedKeys.delete(key);
  }

  getServerForKey(key) {
    // Consistent hashing (simplified)
    const hash = this.hash(key);
    const index = hash % this.cacheServers.length;
    return this.cacheServers[index];
  }

  hash(key) {
    let hash = 0;
    for (let i = 0; i < key.length; i++) {
      hash = ((hash << 5) - hash) + key.charCodeAt(i);
      hash = hash & hash;
    }
    return Math.abs(hash);
  }
}

// Simulation
const cache = new CacheWithHotspotHandling([
  { id: 'cache-1', get: async (k) => `value-${k}`, set: async (k,v) => {}, delete: async (k) => {} },
  { id: 'cache-2', get: async (k) => `value-${k}`, set: async (k,v) => {}, delete: async (k) => {} },
  { id: 'cache-3', get: async (k) => `value-${k}`, set: async (k,v) => {}, delete: async (k) => {} }
]);

// Simulate viral content
```

```
console.log('Simulating viral tweet...');
for (let i = 0; i < 2000; i++) {
  await cache.get('tweet:viral-123');  // Same key, high volume
}

// Output:
// Normal tweet:viral-123 → Server cache-2 (first 1000)
// 🔥 HOTSPOT DETECTED: tweet:viral-123 (1000 req/s)
// Replicating hotspot key: tweet:viral-123
// ✓ Key tweet:viral-123 replicated to all 3 servers
// Hotspot tweet:viral-123 → Random server cache-1 (load balanced!)
// Hotspot tweet:viral-123 → Random server cache-3 (load balanced!)
```

Geographic Data Locality

Keep data close to users geographically.

Global Architecture:



Lookup:

1. US user → Check US cache (5ms)
2. If miss → Check global cache (50ms)
3. If miss → Database (100ms)

EU user → Check EU cache (5ms)

Asia user → Check Asia cache (5ms)

Benefit: 20x lower latency than cross-region

Implementation:

javascript

```
class GeographicCache {
  constructor(region) {
    this.region = region;

    // Local region cache (fast)
    this.localCache = redis.createClient({
      host: `cache-${region}`,
      port: 6379
    });

    // Global cache (slower, fallback)
    this.globalCache = redis.createClient({
      host: 'cache-global',
      port: 6379
    });
  }

  async get(key) {
    const start = Date.now();

    try {
      // Try local cache first
      const local = await this.localCache.get(key);
      if (local) {
        console.log(`Local cache HIT (${this.region}): ${Date.now() - start}ms`);
        return JSON.parse(local);
      }

      // Try global cache
      const global = await this.globalCache.get(key);
      if (global) {
        console.log(`Global cache HIT: ${Date.now() - start}ms`);

        const parsed = JSON.parse(global);

        // Populate local cache
        await this.localCache.setex(key, 3600, global);

        return parsed;
      }

      // Database fallback
      const data = await database.get(key);
      console.log(`Database HIT: ${Date.now() - start}ms`);

      if (data) {
```

```
      // Populate both caches
      const serialized = JSON.stringify(data);
      await this.localCache.setex(key, 3600, serialized);
      await this.globalCache.setex(key, 3600, serialized);
    }

    return data;

  } catch (error) {
    console.error('Cache error:', error);
    return await database.get(key);
  }
}

async set(key, value) {
  // Write to database
  await database.set(key, value);

  const serialized = JSON.stringify(value);

  // Update local cache
  await this.localCache.setex(key, 3600, serialized);

  // Update global cache (async, don't wait)
  this.globalCache.setex(key, 3600, serialized).catch(err => {
    console.error('Failed to update global cache:', err);
  });
}

async invalidate(key) {
  // Invalidate local
  await this.localCache.del(key);

  // Invalidate global
  await this.globalCache.del(key);

  // Optionally: Broadcast to other regions
  await this.broadcastInvalidation(key);
}

async broadcastInvalidation(key) {
  // Publish invalidation message
  await pubsub.publish('cache-invalidation', JSON.stringify({
    key,
    region: this.region,
    timestamp: Date.now()
  }));
```

```
  }
}

// Different regions
const usCache = new GeographicCache('us-east');
const euCache = new GeographicCache('eu-west');
const apCache = new GeographicCache('ap-southeast');

// US user
const user1 = await usCache.get('user:123');  // 5ms (local)

// EU user (same data)
const user2 = await euCache.get('user:123');  // 50ms (global cache)
// But second request: 5ms (now in local cache)
```

Cache Stampede Prevention

Problem: Cache expires, many requests hit database simultaneously.

```
Cache Stampede Scenario:

Popular item cached (1M requests/hour)
Cache expires at 10:00:00

10:00:00.000 - Request 1 arrives
        Cache miss → Query DB
10:00:00.001 - Request 2 arrives
        Cache miss → Query DB
10:00:00.002 - Request 3 arrives
        Cache miss → Query DB
...
10:00:00.100 - Request 100 arrives
        Cache miss → Query DB

100 simultaneous DB queries!
Database overloaded! 💥
```

Solution: Mutex/Lock Pattern

```
javascript
```

```
class CacheWithStampedeProtection {
  constructor(cache, db) {
    this.cache = cache;
    this.db = db;
    this.locks = new Map();
  }

  async get(key) {
    // Check cache
    const cached = await this.cache.get(key);
    if (cached) {
      return JSON.parse(cached);
    }

    // Cache miss - acquire lock
    const lock = await this.acquireLock(key);

    if (lock.acquired) {
      try {
        // Double-check cache (another thread might have loaded it)
        const recheck = await this.cache.get(key);
        if (recheck) {
          return JSON.parse(recheck);
        }

        // Load from database (only one thread does this!)
        console.log(`[LOCK HOLDER] Loading ${key} from database...`);
        const data = await this.db.get(key);

        // Populate cache
        await this.cache.setex(key, 3600, JSON.stringify(data));

        return data;

      } finally {
        // Release lock
        await this.releaseLock(key, lock.token);
      }

    } else {
      // Another thread is loading, wait and retry
      console.log(`[WAITING] Another thread loading ${key}...`);

      // Wait a bit
      await new Promise(resolve => setTimeout(resolve, 100));
```

```
      // Retry (cache should be populated now)
      return await this.get(key);
    }
  }

  async acquireLock(key, timeout = 10000) {
    const lockKey = `lock:${key}`;
    const token = Math.random().toString(36);

    // Try to acquire lock (SET NX - set if not exists)
    const acquired = await this.cache.set(
      lockKey,
      token,
      'PX', timeout,  // Expire after timeout
      'NX'  // Only if not exists
    );

    return {
      acquired: acquired === 'OK',
      token: token
    };
  }

  async releaseLock(key, token) {
    const lockKey = `lock:${key}`;

    // Lua script for atomic check-and-delete
    const script = `
      if redis.call("get", KEYS[1]) == ARGV[1] then
        return redis.call("del", KEYS[1])
      else
        return 0
      end
    `;

    await this.cache.eval(script, 1, lockKey, token);
  }
}

// Usage
const cache = new CacheWithStampedeProtection(redisClient, database);

// Simulate 100 simultaneous requests
const promises = [];
for (let i = 0; i < 100; i++) {
  promises.push(cache.get('popular-item-123'));
}
```

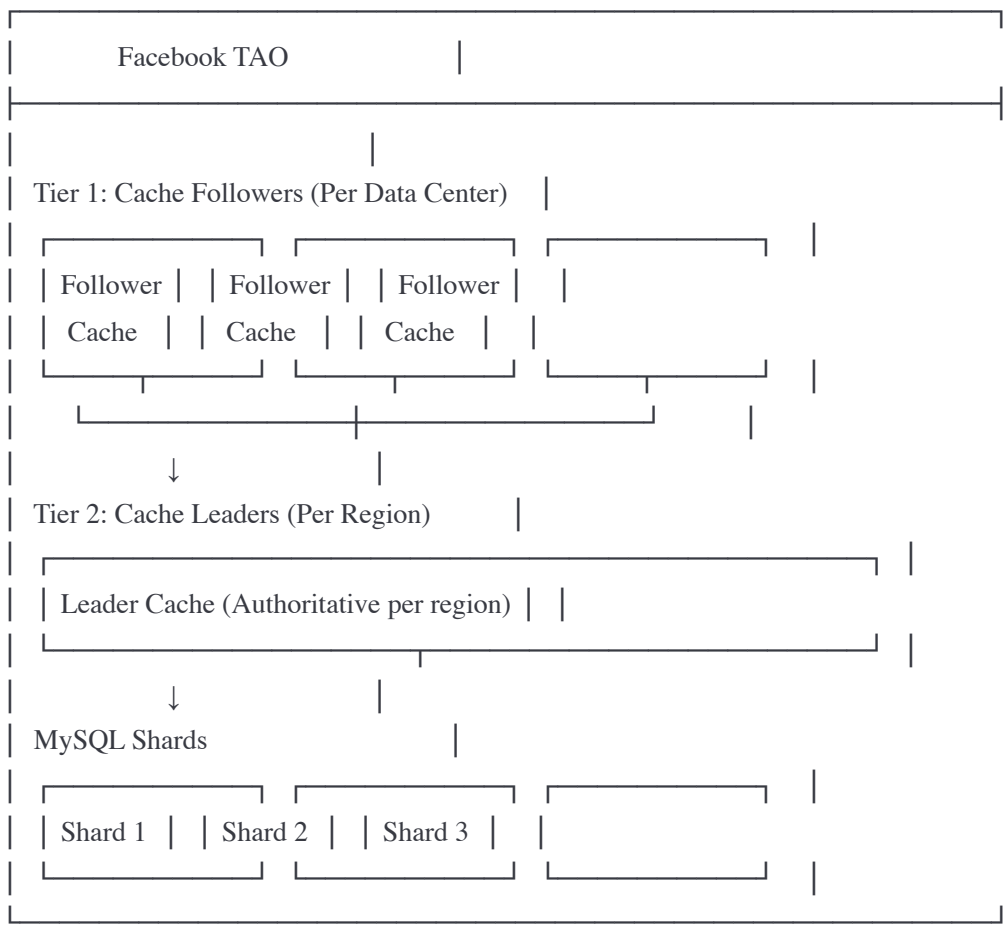
```
await Promise.all(promises);

// Output:
// [LOCK HOLDER] Loading popular-item-123 from database...
// [WAITING] Another thread loading popular-item-123...
// [WAITING] Another thread loading popular-item-123...
// ... (99 threads wait)
// All threads get the value
// Only ONE database query! ✓
```

Real-World Examples

Facebook's TAO (The Associations and Objects)

Facebook's Distributed Cache:

Architecture:



Write Flow:

1. Write to leader cache
2. Write to MySQL
3. Invalidate followers
4. Followers refill from leader

Read Flow:
1. Check follower cache (99% hit rate, 1ms)
2. If miss, check leader cache (1ms)
3. If miss, query MySQL (10ms)

Scale: Billions of requests/second

Twitter's Manhattan

Twitter's Distributed Key-Value Store:

Features:
- Strong consistency within datacenter
- Eventual consistency across datacenters
- Multi-tier caching
- Automatic hotspot detection

Hotspot Handling:
1. Detect hot keys (>10K req/s)
2. Replicate to multiple cache servers
3. Load balance across replicas
4. Automatically de-replicate when cool

Example: Trending tweet
- Normal tweet: 100 req/s → 1 cache server
- Viral tweet: 100K req/s → Replicated to 10 servers
- Each handles 10K req/s ✓

Key Takeaways

1. **Distributed Cache Coherence:**
   - Shared cache (Redis cluster) - simple, consistent
   - Invalidation messages - local cache + notifications
   - Consistent hashing - partition data
   - Multi-tier - balance speed and consistency
2. **Write-Through vs Write-Behind:**
   - Write-through: Sync, slower, consistent
   - Write-behind: Async, faster, eventual consistency

- Batching for efficiency
- Choose based on requirements

3. **Session Management:**
   - Sticky sessions - simple, poor scaling
   - Shared session store - scalable, Redis dependency
   - JWT - stateless, no storage needed
   - Hybrid approaches
4. **Data Locality:**
   - Multi-tier caching (L1 local, L2 shared)
   - Geographic distribution
   - Bring data close to computation
   - 10-100x latency improvement
5. **Hotspot Management:**
   - Detect hot keys
   - Replicate to multiple servers
   - Load balance across replicas
   - Mutex for cache stampede
   - Auto-scale for viral content

Practice Problems

1. Design a caching layer for Instagram (global users, viral posts). How would you handle hotspots?
2. Calculate: L1 cache (0.1ms, 90% hit), L2 cache (1ms, 9% hit), DB (10ms, 1% hit). Average latency?
3. Design session management for a banking app. Which approach and why?

Congratulations! 🎉

You've completed **19 comprehensive chapters!** This is excellent progress covering all major system design topics.

Would you like to:

1. **Practice complete system design problems** (highly recommended now!)
2. Continue with remaining specialized chapters
3. Deep dive on specific topics

What interests you most?