# Caching Strategies and Implementation

*Author: Chandan Kumar*

July 19, 2025

## Contents

# 1 Introduction to Caching

## 1.1 What is Caching?

Caching is a technique used to store frequently accessed data in a temporary storage location (cache) that provides faster access than the original data source. The primary goal is to reduce latency, improve performance, and decrease load on backend systems.

**Key Point**

Cache hit rate is a critical metric: Hit Rate $= \frac{\text{Cache Hits}}{\text{Total Requests}} \times 100\%$

## 1.2 Why Caching Matters

- **Performance**: Reduces response time from milliseconds to microseconds
- **Scalability**: Reduces load on databases and backend services
- **Cost Efficiency**: Decreases computational resources and bandwidth usage
- **User Experience**: Faster page loads and smoother interactions

## 1.3 Cache Hierarchy

1. **Browser Cache**: Client-side caching (HTML, CSS, JS, images)
2. **CDN Cache**: Geographic distribution of static content
3. **Application Cache**: In-memory caching within applications
4. **Database Cache**: Query result caching

# 2 Cache Replacement Strategies

## 2.1 Least Recently Used (LRU)

Removes the least recently accessed item when cache is full.

**Algorithm 1** LRU Cache Implementation

```
Initialize doubly linked list and hash map Getkey
if key exists in hash map then
    Move node to front of list
    return node.value
else
    return -1
end ifPutkey, value
if key exists then
    Update value and move to front
else
    if cache is full then
        Remove tail node
    end if
    Add new node to front
end if
```

- **Time Complexity**: O(1) for both get and put operations
- **Space Complexity**: O(capacity)

**Example**

LRU with capacity 3:

- Put(1,1), Put(2,2), Put(3,3) → Cache: [3,2,1]
- Get(2) → Cache: [2,3,1]
- Put(4,4) → Cache: [4,2,3] (1 evicted)

### 2.2 Least Frequently Used (LFU)

Removes the item with the lowest access frequency.

- Maintain frequency counter for each key
- Use min-heap or sorted structure for efficient eviction
- Handle tie-breaking with recency information

### 2.3 First In First Out (FIFO)

Removes the oldest item regardless of usage patterns.

- Simple to implement using queue
- O(1) insertion and deletion
- Doesn't consider access patterns
- Lower hit rates compared to LRU/LFU

## 3 Caching Patterns

### 3.1 Cache-Aside (Lazy Loading)

Application manages cache explicitly.

Listing 1: Cache-Aside Pattern

```
def get_user(user_id):
    # Try cache first
    user = cache.get(f"user:{user_id}")
    if user is None:
        # Cache miss - fetch from database
        user = database.get_user(user_id)
        # Store in cache for future requests
        cache.set(f"user:{user_id}", user, ttl=3600)
    return user

def update_user(user_id, user_data):
    # Update database
    database.update_user(user_id, user_data)
    # Invalidate cache
    cache.delete(f"user:{user_id}")
```

- **Pros**: Application controls caching logic, handles cache misses gracefully
- **Cons**: Risk of stale data, cache miss penalty

### 3.2 Write-Through

Data is written to cache and database simultaneously.

Listing 2: Write-Through Pattern

```
def update_user(user_id, user_data):
    # Write to database first
    database.update_user(user_id, user_data)
    # Update cache
    cache.set(f"user:{user_id}", user_data, ttl=3600)
    return user_data
```

- **Pros**: Data consistency, cache always up-to-date
- **Cons**: Higher write latency, cache pollution

### 3.3 Write-Behind (Write-Back)

Data written to cache immediately, database updated asynchronously.

**Pros**: Low write latency, better performance

**Cons**: Risk of data loss, complex error handling

### 3.4 Refresh-Ahead

Proactively refresh cache before expiration.

Listing 3: Refresh-Ahead Implementation

```
def get_user_with_refresh(user_id):
    user, timestamp = cache.get_with_timestamp(f"user:{user_id}")

    if user is None:
        # Cache miss
        return load_and_cache_user(user_id)

    # Check if refresh needed (e.g., 80% of TTL elapsed)
    if time.now() - timestamp > 0.8 * TTL:
        # Trigger async refresh
        background_refresh(user_id)

    return user
```

## 4 Distributed Caching

### 4.1 Consistent Hashing

Distributes cache keys across multiple nodes while minimizing redistribution during node changes.

**Key Point**

Hash Ring: Keys and nodes mapped to points on a circle. Each key stored on the first node clockwise from its position.

**Advantages**:

- Minimal key redistribution when nodes join/leave
- Load balancing across nodes
- Fault tolerance

### 4.2 Cache Coherence

Ensuring consistency across distributed cache nodes.

1. **Invalidation**: Remove stale data from all nodes
2. **Update Propagation**: Broadcast updates to all nodes
3. **Time-based Expiration**: Use TTL for automatic cleanup

## 5 Advanced Caching Concepts

### 5.1 Cache Stampede

Multiple requests for the same missing key simultaneously hit the backend.

- **Locking**: First request locks the key during fetch
- **Probabilistic Early Expiration**: Randomly expire before TTL
- **Background Refresh**: Update cache asynchronously

Listing 4: Cache Stampede Prevention

```
def get_with_lock(key):
    value = cache.get(key)
    if value is None:
        with distributed_lock(f"lock:{key}"):
            # Double-check after acquiring lock
            value = cache.get(key)
            if value is None:
                value = expensive_operation(key)
                cache.set(key, value, ttl=3600)
    return value
```

### 5.2 Multi-Level Caching

Hierarchical cache structure with different characteristics.

1. **L1**: In-process cache (fastest, smallest)
2. **L2**: Distributed cache (Redis/Memcached)
3. **L3**: Database query cache

### 5.3 Cache Warming

Pre-populating cache with anticipated data.

- **Batch Loading**: Load popular items during off-peak hours
- **Predictive Loading**: Use analytics to predict access patterns
- **User-Triggered**: Warm cache based on user behavior

# 6 Performance Metrics

## 6.1 Key Metrics

$$\text{Hit Rate} = \frac{\text{Cache Hits}}{\text{Total Requests}} \tag{1}$$

$$\text{Miss Rate} = 1 - \text{Hit Rate} \tag{2}$$

$$\text{Effective Access Time} = \text{Hit Rate} \times T_{\text{cache}} + \text{Miss Rate} \times T_{\text{backend}} \tag{3}$$

## 6.2 Monitoring and Optimization

- **Hit/Miss Ratios**: Monitor per endpoint and overall
- **Eviction Rates**: High eviction indicates undersized cache
- **Memory Usage**: Track cache memory consumption
- **Latency Metrics**: 95th/99th percentile response times

# 7 Common Interview Questions

**Interview Question**

**Q1**: Implement an LRU cache with O(1) get and put operations.
**Approach**: Use HashMap + Doubly Linked List

- HashMap for O(1) key lookup
- Doubly linked list for O(1) insertion/deletion
- Move accessed nodes to head
- Remove tail when capacity exceeded

**Interview Question**

**Q2**: How would you design a distributed cache for a social media application?
**Key Considerations**:

- User data partitioning strategy
- Consistency requirements
- Cache invalidation on updates
- Handling hot keys (celebrity users)
- Geographic distribution (CDN)

**Interview Question**

**Q3**: Explain cache stampede and how to prevent it.
**Answer**: Multiple concurrent requests for expired/missing cache key overwhelm backend. Solutions include distributed locking, probabilistic expiration, and background refresh patterns.

# 8 Best Practices

## 8.1 Cache Design Principles

1. **Cache What's Expensive**: Focus on computationally expensive or slow database queries
2. **Set Appropriate TTL**: Balance between freshness and performance
3. **Handle Cache Misses Gracefully**: Always have fallback to original data source
4. **Monitor Cache Performance**: Track hit rates, memory usage, and latency
5. **Plan for Cache Failures**: Design system to work without cache

## 8.2 Common Pitfalls

- **Cache Everything**: Over-caching can waste memory and complicate invalidation
- **Ignore Consistency**: Not handling cache invalidation properly
- **Cache Hot Spots**: Not distributing load for popular keys
- **No Monitoring**: Not tracking cache effectiveness
- **Complex Cache Keys**: Making debugging and maintenance difficult

# 9 Conclusion

Caching is a fundamental technique for building scalable, high-performance applications. Understanding different caching strategies, implementation patterns, and trade-offs is crucial for SDE-1 interviews and real-world system design.

**Key Point**

Remember: Caching introduces complexity through consistency challenges, but the performance benefits often justify this trade-off in distributed systems.

**Key Takeaways**:

- Choose appropriate cache replacement policy based on access patterns
- Understand trade-offs between different caching patterns
- Design for cache failures and implement proper monitoring
- Consider distributed caching challenges in system design interviews

**Interview Question**

**Q4**: You have a cache with 80% hit rate and 100ms backend latency. Cache latency is 1ms. If you double the cache size and hit rate becomes 90%, what's the performance improvement?
**Solution**:

$$\text{Original EAT} = 0.8 \times 1 + 0.2 \times 100 = 20.8\text{ms} \tag{4}$$

$$\text{New EAT} = 0.9 \times 1 + 0.1 \times 100 = 10.9\text{ms} \tag{5}$$

$$\text{Improvement} = \frac{20.8 - 10.9}{20.8} = 47.6\% \tag{6}$$

Interview Question

**Q5**: Design a cache that supports both TTL-based and LRU eviction. How would you implement this hybrid approach?
**Key Points**:

- Maintain both timestamp and LRU ordering
- Background thread for TTL cleanup
- Lazy deletion during get operations
- Priority: TTL expiration ¿ LRU eviction

Interview Question

**Q6**: Your application caches user profiles. During peak hours, you notice cache hit rate drops from 95% to 60%. What could be the reasons and solutions?
**Analysis**:

- **Cause**: Cache eviction due to memory pressure, new user registrations
- **Solutions**: Increase cache size, implement cache warming, use probability-based eviction
- **Monitoring**: Track eviction rate, memory usage patterns

Interview Question

**Q7**: Implement a write-through cache that handles concurrent writes to the same key safely.
**Approach**:

- Use distributed locks or compare-and-swap operations
- Implement versioning for conflict detection
- Handle partial failures (DB success, cache failure)
- Consider using write coalescing for high-frequency updates

Interview Question

**Q8**: You're designing a cache for a recommendation system. The cache needs to handle 1M+ keys with different access patterns. How would you optimize it?
**Strategy**:

- **Tiered Caching**: Hot data in memory, warm data in Redis
- **Adaptive TTL**: Shorter TTL for rapidly changing recommendations
- **Batch Updates**: Group recommendation updates
- **Probabilistic Structures**: Use Bloom filters for negative caching

Interview Question

**Q9**: Explain the thundering herd problem in caching and implement a solution using Redis.
**Problem**: When a popular cache key expires, multiple threads simultaneously fetch from DB, overloading it.
**Solution**: Use Redis locks to ensure only one thread fetches and populates the cache, while others wait.
**Python Code Using Redis Lock**:

```
def get_with_lock(key, fetch_func, ttl=3600):
    # Try to get cached value
    value = redis.get(key)
    if value:
        return json.loads(value)

    # Try to acquire lock
    lock_key = f"lock:{key}"
    if redis.set(lock_key, "1", nx=True, ex=30):
        try:
            # Double check after lock
            value = redis.get(key)
            if not value:
                value = fetch_func()
                redis.setex(key, ttl, json.dumps(value))
            return json.loads(value) if value else None
        finally:
            redis.delete(lock_key)
    else:
        # Wait and retry
        time.sleep(0.1)
        return get_with_lock(key, fetch_func, ttl)
```

Interview Question

**Q10**: Design a cache system that can handle cache warming for 1 billion user profiles efficiently.
**Architecture**:

- **Batch Processing**: Process users in chunks of 10K
- **Priority Queue**: Warm active users first
- **Distributed Workers**: Multiple warming services
- **Rate Limiting**: Prevent overwhelming the database
- **Checkpointing**: Resume warming after failures

Interview Question

**Q11**: You notice that 20% of your cache keys consume 80% of the memory. How would you handle this "hot key" problem?
**Solutions**:

- **Key Sharding**: Split large values across multiple keys
- **Compression**: Compress large values before caching
- **Separate Tier**: Use different cache policies for hot keys
- **Monitoring**: Track key size distribution
- **Client-side Caching**: Cache hot keys at application level

Interview Question

**Q12**: Implement a cache that supports atomic increment operations with TTL refresh.
**Challenge**: Increment counter and refresh TTL atomically.
**Redis Lua Script Solution**:

```
-- KEYS[1]: counter key
-- ARGV[1]: increment value
-- ARGV[2]: TTL in seconds
local current = redis.call('GET', KEYS[1])
if current == false then
    current = 0
else
    current = tonumber(current)
end
local new_value = current + tonumber(ARGV[1])
redis.call('SETEX', KEYS[1], ARGV[2], new_value)
return new_value
```