Caching in System Design - HLD Interview Guide

Table of Contents

What is Caching?

Definition

Caching is storing copies of frequently accessed data in a faster storage layer to reduce access time and computational overhead.

Core Concept

```
Without Cache:
  Request → Database (slow, ~50-100ms)

With Cache:
  Request → Cache (fast, ~1ms) ✓
      ↓ (cache miss)
  Database (slow, ~50-100ms)
```

Key Principles

1. Locality of Reference

```
Temporal Locality:
  Recently accessed items likely accessed again
  Example: User views profile multiple times in session

Spatial Locality:
```

```
  Items near recently accessed items likely accessed
  Example: If user reads post #100, likely to read #101, #102
```

2. The 80/20 Rule (Pareto Principle)

```
80% of requests hit 20% of data
→ Cache that 20% for massive gains

Example:
  Total products: 1M
  Popular products: 200K (20%)
  If cached: 80% requests served from cache
```

3. Cost vs. Benefit

```
Cost:
├── Memory (cache storage)
├── Complexity (invalidation logic)
└── Stale data risk

Benefit:
├── 10-100x faster response times
├── Reduced database load
├── Lower operational costs
└── Better user experience
```

Why Caching Matters

Performance Impact

```
E-commerce Product Page Example:

Without Cache:
├── Database query: 50ms
├── Image processing: 100ms
├── Recommendations: 200ms
├── Total: 350ms
└── Cost: $0.01 per request

With Cache:
├── Cached HTML: 1ms
├── Cached images: 1ms (CDN)
├── Cached recommendations: 1ms
```

```
├── Total: 3ms (117x faster!)
└── Cost: $0.0001 per request
```

Scalability Impact

```
Database Without Cache:
├── Max throughput: 1,000 QPS
├── Need 10 servers for 10,000 QPS
└── Cost: $10,000/month

With Cache (95% hit rate):
├── Cache handles: 9,500 QPS
├── Database handles: 500 QPS
├── Need 1 server for 10,000 QPS
└── Cost: $1,500/month (85% savings!)
```

When NOT to Cache

```
❌ Frequently changing data (< 1 second lifespan)
❌ Large datasets with low hit rate (<10%)
❌ Strict consistency required (financial data)
❌ Highly personalized, no reuse
❌ Cheaper to compute than store
```

Cache Hierarchy & Layers

Complete Cache Stack

```
┌─────────────────────────────────────────────┐
│ Layer 1: CLIENT-SIDE CACHE                  │
│ Browser cache, Mobile app cache             │
│ Latency: 0ms (local)                        │
│ Hit Rate: 30-50%                            │
└─────────────────────────────────────────────┘
        ↓ (miss)
┌─────────────────────────────────────────────┐
│ Layer 2: CDN / EDGE CACHE                   │
│ CloudFront, Cloudflare, Akamai              │
│ Latency: 10-50ms (edge location)            │
│ Hit Rate: 70-90% (static content)           │
└─────────────────────────────────────────────┘
        ↓ (miss)
┌─────────────────────────────────────────────┐
│ Layer 3: APPLICATION CACHE (In-Memory)      │
```

```
│ Caffeine, Guava, Local cache                          │
│ Latency: 0.1-1ms                                      │
│ Hit Rate: 40-60%                                      │
└──────────────────────────────────────────────────────┘
        ↓ (miss)
┌──────────────────────────────────────────────────────┐
│ Layer 4: DISTRIBUTED CACHE                            │
│ Redis, Memcached                                      │
│ Latency: 1-5ms (network call)                         │
│ Hit Rate: 80-95%                                      │
└──────────────────────────────────────────────────────┘
        ↓ (miss)
┌──────────────────────────────────────────────────────┐
│ Layer 5: DATABASE CACHE                               │
│ Query cache, Buffer pool (MySQL InnoDB)               │
│ Latency: 5-20ms                                       │
│ Hit Rate: 60-80%                                      │
└──────────────────────────────────────────────────────┘
        ↓ (miss)
┌──────────────────────────────────────────────────────┐
│ Layer 6: DISK STORAGE                                 │
│ Latency: 50-100ms (SSD), 5-15ms (NVMe)                │
└──────────────────────────────────────────────────────┘
```

Layer Details

1. Client-Side Cache (Browser/Mobile)

```
Browser Cache Types:
├── HTTP Cache (Cache-Control headers)
├── Service Workers (offline support)
├── LocalStorage (5-10MB persistent)
└── SessionStorage (tab-specific)

Configuration Example:
  Cache-Control: max-age=3600, public
  ETag: "33a64df551425fcc55e"

Use Cases:
✓ Static assets (CSS, JS, images)
✓ API responses (short TTL)
✓ User preferences
✓ Form data (temporary)
```

2. CDN / Edge Cache

```
How CDN Works:

User in Tokyo requests: example.com/logo.png
  ↓
DNS resolves to nearest edge (Tokyo)
  ↓
Edge cache checks:
  ├── HIT: Return image (10ms)
  └── MISS: Fetch from origin (200ms), cache it

Next Tokyo user: Served in 10ms

Characteristics:
├── 200+ global edge locations
├── Automatic geographic routing
├── 85-95% hit rate for static content
└── Massive bandwidth savings
```

CDN Cache Configuration

```
Cache Behaviors by Path:
├── /static/*  → Cache 1 year (immutable)
├── /api/*     → Cache 5 minutes (dynamic)
├── /images/*  → Cache 1 day
└── /          → Cache 1 hour (HTML)

Cache Key Components:
├── URL path
├── Query strings
├── Headers (Accept-Language)
└── Cookies (personalization)
```

3. Application Cache (In-Memory)

```
Configuration:
├── Max size: 10,000 entries
├── TTL: 10 minutes
├── Eviction: LRU policy
├── Statistics: Hit/miss rates

Characteristics:
├── Fastest (no network)
├── Limited to single server
├── Lost on restart
└── Perfect for: Config, sessions

Use Cases:
```

```
✓ Application configuration
✓ User sessions (small scale)
✓ Frequently accessed objects
✓ Computed results
✓ Rate limiting counters
```

4. Distributed Cache (Redis/Memcached)

```
Architecture:

┌─────────────────────────────────────────────┐
│    Application Servers (10+)                │
│  App1  App2  App3  ...  App10               │
└──┬─────┬─────┬──────────┬───────────────────┘
   │     │     │          │
   └─────┴─────┴──────────┘
                 │
                 ↓
┌─────────────────────────────────────────────┐
│      Redis Cluster (Shared)                 │
│  ┌──────────┐  ┌──────────┐  ┌──────────┐   │
│  │Shard 1 │  │Shard 2 │  │Shard 3 │   │
│  └──────────┘  └──────────┘  └──────────┘   │
└─────────────────────────────────────────────┘

Benefits:
✓ Shared across all servers
✓ Survives server restarts
✓ Horizontal scaling
✓ Sub-millisecond latency
```

Redis Data Structures

```
1. String: Simple key-value
   user:123 → {"name":"John","age":30}

2. Hash: Structured data
   user:123 → {name: "John", age: 30}

3. List: Ordered collections
   recent_views:user123 → [prod1, prod2, prod3]

4. Set: Unique items
   user:123:friends → {user456, user789}

5. Sorted Set: Rankings/leaderboards
   leaderboard → {user123: 1500, user456: 1200}
```

5. Database Cache

```
Query Result Cache (MySQL 8.0 deprecated):
  Problems:
  ❌ Invalidates entire cache on ANY write
  ❌ Not suitable for write-heavy workloads

Buffer Pool (InnoDB):
  ✓ Automatic caching of data pages
  ✓ Recommended: 70-80% of RAM
  ✓ LRU eviction
  ✓ Transparent to application

Configuration:
  buffer_pool_size = 8GB
  Significantly faster repeated queries
```

Caching Strategies

1. Cache-Aside (Lazy Loading) ⭐ Most Common

```
Read Flow:
  1. Application checks cache
  2. Cache HIT → Return data (1ms)
  3. Cache MISS:
     - Query database (50ms)
     - Store in cache with TTL
     - Return data

Write Flow:
  1. Update database
  2. Invalidate cache
  3. Next read repopulates cache
```

When to Use

```
✓ Read-heavy (10:1 read:write)
✓ Data changes infrequently
✓ Can tolerate cache misses
✓ Fine-grained control needed

Examples:
- User profiles
- Product catalogs
- Blog posts
- Configuration
```

Pros & Cons

```
Pros:
✓ Only caches requested data
✓ Cache failure doesn't break system
✓ Simple to implement
✓ Good for read-heavy loads

Cons:
✗ Cache miss penalty (3 network calls)
✗ Stale data risk
✗ Cache stampede vulnerability
```

2. Read-Through Cache

```
Read Flow:
  1. Application queries cache
  2. Cache checks data:
     - HIT: Return data
     - MISS: Cache loads from DB → stores → returns

Key Difference: Cache manages DB loading, not application

When to Use:
✓ Want simpler application logic
✓ Consistent caching behavior
✓ Cache library available
```

3. Write-Through Cache

```
Write Flow:
  1. Application writes to cache
  2. Cache synchronously writes to database
  3. Both updated before response

Read Flow:
  1. Read from cache (always present)

Result: Cache and database always synchronized

When to Use:
✓ Need strong consistency
✓ Read-after-write pattern
✓ Can tolerate higher write latency
```

```
Examples:
- User preferences
- Shopping cart
- Session data
```

4. Write-Back (Write-Behind) Cache

```
Write Flow:
  1. Application writes to cache (1ms) ✓
  2. Cache acknowledges immediately
  3. Cache queues async database write
  4. Background worker persists to DB

Read Flow:
  1. Read from cache (has latest, including pending writes)

When to Use:
✓ Write-heavy workload
✓ Can tolerate eventual consistency
✓ Write performance critical

⚠ Risk: Data loss if cache crashes before DB write

Examples:
- Analytics events
- Logging
- Metrics

❌ NOT for: Financial data, critical transactions
```

5. Write-Around Cache

```
Write Flow:
  1. Write directly to database (bypass cache)
  2. Invalidate cache if exists
  3. Cache repopulated on next read

When to Use:
✓ Write-once, read-rarely data
✓ Avoid cache pollution
✓ Write performance critical

Examples:
- Log files
- Archive data
- Batch uploads
```

6. Refresh-Ahead Cache

```
Mechanism:
  1. Monitor cached items approaching expiry
  2. At 75% of TTL: Trigger async refresh
  3. Update cache before actual expiry
  4. Users always get cache hits

Example Timeline:
  T=0s:  Item cached (TTL=60s)
  T=45s: Async refresh triggered
  T=46s: Cache updated
  T=50s: User request → HIT (fast!)

When to Use:
✓ Latency-sensitive APIs
✓ Predictable access patterns
✓ Cache misses unacceptable

Examples:
- Home page content
- Top products
- Trending topics
```

Cache Eviction Policies

Why Eviction Matters

```
Problem: Limited memory

Cache: 1GB
Total Data: 100GB
Must decide: Which 1% to keep?

Bad eviction → Low hit rate → Poor performance
Good eviction → High hit rate → Fast system
```

1. LRU (Least Recently Used) ⭐ Recommended

```
Algorithm: Evict least recently accessed item

Structure: Doubly Linked List + HashMap

┌─────────────────────────────────────────┐
│  MRU (Most Recently Used)               │
```

```
┌──────────────────────────────────────────┐
│  ↓                                       │
│  [Item E] ↔ [Item C] ↔ [Item A]          │
│  ↓                                       │
│  LRU (Least Recently Used) ← EVICT       │
└──────────────────────────────────────────┘

Operations:
- Access: Move to front (MRU)
- Add: Insert at front
- Evict: Remove from back (LRU)
- All O(1) time complexity

Characteristics:
✓ Simple and intuitive
✓ Good for temporal locality
✓ O(1) operations
✓ Most widely used

Best For:
- General purpose caching
- Web applications
- Database query cache
```

Example Scenario

```
Cache: [A, B, C] (capacity=3)

Access A: [A, B, C] (A moves to front)
Access D: [D, A, B] (C evicted)
Access B: [B, D, A] (B moves to front)
Access E: [E, B, D] (A evicted)

Limitation: Recent ≠ Frequent
If A accessed 1000x, then B, C, D once
A still gets evicted!
```

2. LFU (Least Frequently Used)

```
Algorithm: Evict item with lowest access count

Structure: Frequency Buckets

┌──────────────────────────────────────────┐
│  Freq 5: [Item A]                        │
│  Freq 3: [Item B] [Item C]               │
│  Freq 1: [Item D] [Item E] ← EVICT       │
└──────────────────────────────────────────┘
```

```
Characteristics:
✓ Keeps truly popular items
✓ Resistant to scans
✗ Complex implementation
✗ New items disadvantaged

Best For:
- Long-running caches
- Stable access patterns
- Content recommendation
```

3. FIFO (First-In-First-Out)

```
Algorithm: Evict oldest inserted item

Simple queue structure

Characteristics:
✓ Very simple
✗ Ignores access patterns
✗ Poor performance

Generally NOT recommended
```

Comparison

```
┌──────────┬──────────┬────────────┬──────────┐
│ Policy   │ Hit Rate │ Complexity │ Use Case │
├──────────┼──────────┼────────────┼──────────┤
│ LRU      │ High     │ O(1)       │ General  │
│ LFU      │ Higher   │ O(log N)   │ Stable   │
│ FIFO     │ Low      │ O(1)       │ Simple   │
│ Random   │ Lowest   │ O(1)       │ Testing  │
└──────────┴──────────┴────────────┴──────────┘

Recommendation: Start with LRU
```

Cache Consistency & Invalidation

The Challenge

```
Scenario:
T0: Update profile in database
T1: Cache still has old data
```

```
T2: User reads from cache → sees stale data

Duration: Until cache expires (TTL) or invalidated
```

Invalidation Strategies

1. Time-Based (TTL)

```
Mechanism:
  - Set expiry time on cached data
  - Automatic eviction after TTL
  - Simple and predictable

Pros:
✓ Simple to implement
✓ Automatic cleanup
✓ Predictable

Cons:
✗ Data stale until expiry
✗ Doesn't reflect immediate updates
```

TTL Selection Guide

```
Data Type          | TTL        | Reasoning
──────────────────────────────────────────────
Static content     | 1 year     | Rarely changes
Product catalog    | 1 hour     | Infrequent updates
User profiles      | 5–15 min   | Moderate updates
Session data       | Session    | Active usage
Real-time data     | 5–30 sec   | Frequent changes
Auth tokens        | Validity   | Security critical
```

2. Event-Based Invalidation

```
Mechanism:
  - On data update → explicitly invalidate cache
  - Can delete or update cache entry

Workflow:
  1. Update database
  2. Delete/update cache entry
  3. Next read sees fresh data

Pros:
```

```
✓ No stale data
✓ Immediate consistency
✓ Better UX

Cons:
✗ Code coordination needed
✗ Risk of missing invalidations
✗ More complex
```

3. Message Queue Based

```
Architecture:
  Service A → Database → Publish Event → Kafka
                                          ↓
                          Services B, C, D (invalidate cache)

Workflow:
  1. Service updates database
  2. Publishes invalidation event
  3. All services receive event
  4. Each service invalidates own cache

Pros:
✓ Decoupled services
✓ Scalable
✓ Reliable delivery

Cons:
✗ Eventual consistency
✗ Additional infrastructure
✗ Complexity
```

Cache Stampede Problem

The Problem

```
Popular item expires from cache

T0: Item expires
T1: 10,000 concurrent requests arrive
T2: All 10,000 miss cache
T3: All 10,000 query database simultaneously
T4: Database overwhelmed → CRASHES

Called: "Cache Stampede" or "Thundering Herd"
```

Solution 1: Lock-Based

```
Mechanism:
  - Use distributed lock per cache key
  - First request acquires lock, queries DB
  - Others wait, then read from cache

Result:
  ✓ Only 1 database query
  ✓ 9,999 requests served from cache
  ✓ ~500ms total (including wait)

Tools:
  - Redis SETNX for locks
  - ZooKeeper lock recipes
```

Solution 2: Probabilistic Early Expiration

```
Mechanism:
  - Refresh cache BEFORE actual expiry
  - Probability increases as expiry approaches
  - Spreads refreshes over time

Timeline (TTL=60s):
  T=45s: 25% chance to refresh
  T=50s: 50% chance to refresh
  T=55s: 90% chance to refresh

Result:
  ✓ No simultaneous expirations
  ✓ Gradual database load
  ✓ No stampede
```

Solution 3: Serve Stale Data

```
Mechanism:
  - Keep expired data temporarily
  - Serve stale while refreshing async

Workflow:
  1. Cache expires
  2. First request: Return stale + trigger refresh
  3. Others: Get stale data (fast!)
  4. After refresh: Fresh data available

Result:
  ✓ Zero user wait time
  ✓ Only 1 DB query
  ✓ Acceptable staleness (seconds)
```

Distributed Caching

Why Distribute?

```
Single Server Limits:
├── Memory: ~128GB max
├── Single point of failure
├── No horizontal scaling
└── Data lost on restart

Distributed Benefits:
├── Terabytes of memory
├── High availability
├── Fault tolerance
└── Better performance
```

Redis Cluster Architecture

```
┌─────────────────────────────────────────────┐
│         Redis Cluster (16,384 slots)        │
│                                             │
│  Master 1: Slots 0-5460                     │
│    ↓ (replication)                          │
│  Replica 1                                  │
│                                             │
│  Master 2: Slots 5461-10922                 │
│    ↓ (replication)                          │
│  Replica 2                                  │
│                                             │
│  Master 3: Slots 10923-16383                │
│    ↓ (replication)                          │
│  Replica 3                                  │
└─────────────────────────────────────────────┘

Routing:
  slot = CRC16(key) % 16384
  Example: "user:123" → slot 5265 → Master 1
```

Consistent Hashing

Problem with Traditional Hashing

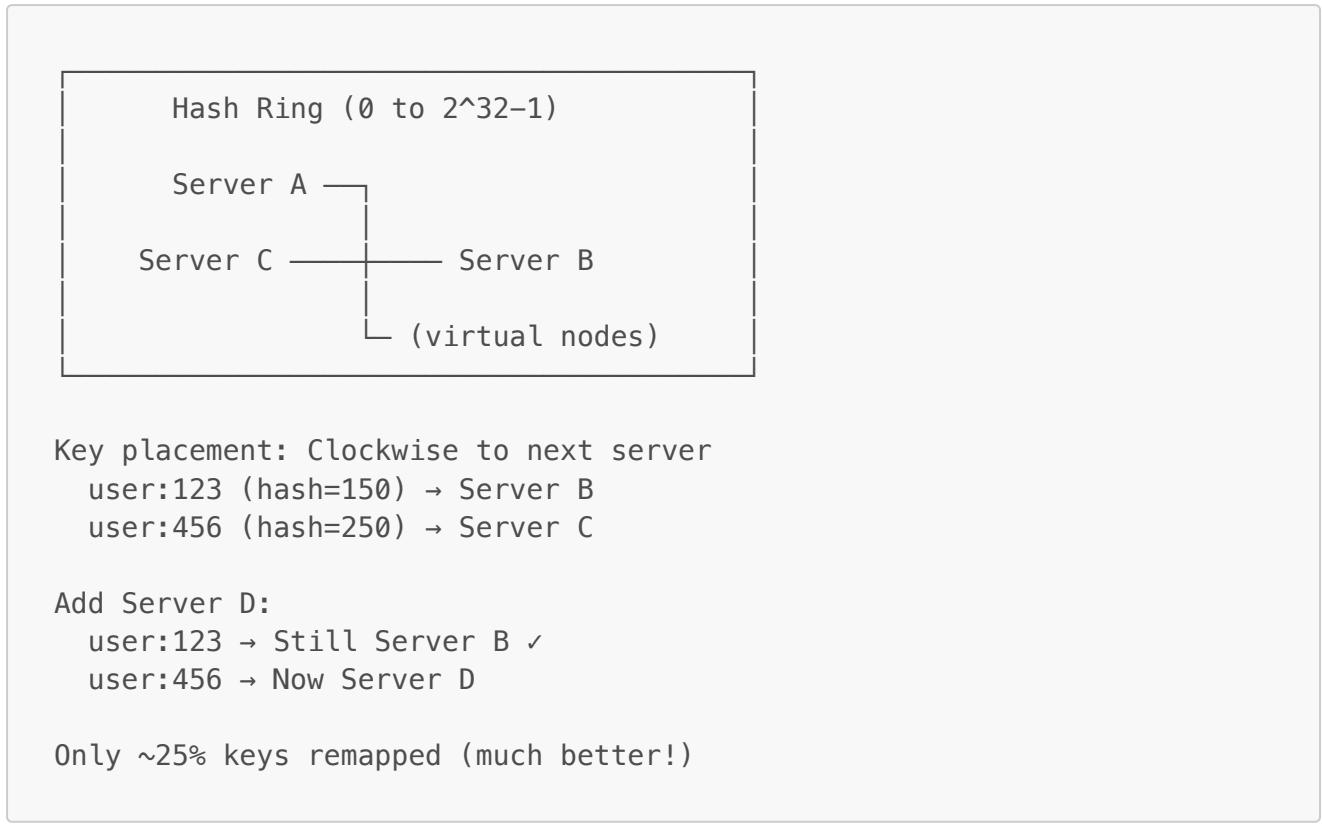
```
server = hash(key) % num_servers
```

```
With 3 servers:
  "user:123" → 12345 % 3 = 0 → Server 0

Add 4th server:
  "user:123" → 12345 % 4 = 1 → Server 1

Result: Different server! Cache miss!
Impact: 75% of 1M cached items remapped!
```

Consistent Hashing Solution




```
Key placement: Clockwise to next server
  user:123 (hash=150) → Server B
  user:456 (hash=250) → Server C

Add Server D:
  user:123 → Still Server B ✓
  user:456 → Now Server D

Only ~25% keys remapped (much better!)
```

Replication Strategy

```
Master-Slave (Recommended):
  Master (Write) → Slave 1, 2, 3 (Read)

  ✓ Fast writes
  ✓ Read scalability
  ✗ Replication lag (1-5ms)
  ✗ Possible data loss

Master-Master:
  Master A ↔ Master B

  ✓ No single point of failure
  ✗ Conflict resolution needed
  ✗ More complex
```

When to Use Caching in HLD Interviews

Decision Framework

✅ Strong Signals to Recommend Caching

1. Read-Heavy Workload (>10:1 ratio)

```
Indicators:
- "Profile viewed 1000x more than updated"
- "Product pages get millions of views"
- "Search results queried repeatedly"

Tell Interviewer:
"This is a read-heavy workload with 100:1 ratio.
Caching can reduce 90% of database load and improve
latency from 50ms to 1ms."
```

2. Expensive Operations

```
Indicators:
- "Requires 5 table joins"
- "ML recommendation takes 200ms"
- "Image transformation needed"
- "Complex aggregation query"

Tell Interviewer:
"These operations are expensive. By caching results,
we reduce computation from 200ms to 1ms and save
99% of processing costs."
```

3. Hot Data (80/20 Rule)

```
Indicators:
- "Top 20% of products → 80% of traffic"
- "Celebrity posts viewed millions of times"
- "Popular searches repeated frequently"

Tell Interviewer:
"Following the 80/20 principle, caching the top 20%
of data will serve 80% of requests. With 1GB cache,
we can achieve 90%+ hit rate."
```

4. Repeated Access

```
Indicators:
- "Users check feed multiple times per hour"
- "Same data requested within minutes"
- "Session data accessed repeatedly"

Tell Interviewer:
"Users exhibit temporal locality. Caching recent
accesses for 5 minutes will capture 85% of requests."
```

5. Static/Slow-Changing Data

```
Indicators:
- "Product descriptions rarely change"
- "Config updated once per day"
- "Historical data (immutable)"

Tell Interviewer:
"This data is static or slow-changing. We can cache
aggressively with long TTL (hours to days) and achieve
95%+ hit rates."
```

6. Performance Critical

```
Indicators:
- "Need sub-100ms response"
- "Serving 100K+ requests/second"
- "Real-time user experience"

Tell Interviewer:
"Performance is critical. Caching enables sub-10ms
responses and allows us to handle 100K QPS with
minimal infrastructure."
```

❌ Red Flags Against Caching

1. Frequently Changing

```
- "Stock prices update every second"
- "Real-time bidding"
- "Live sports scores"

Tell Interviewer:
"This data changes too frequently for effective caching.
The TTL would be <1 second, making cache hit rate too low
to justify the complexity."
```

2. Write-Heavy

```
- "More writes than reads"
- "Data written but rarely read"

Tell Interviewer:
"Write-heavy pattern doesn't benefit from caching.
We'd spend more time invalidating than serving from cache."
```

3. Strong Consistency

```
- "Financial transactions"
- "Exact inventory counts"
- "Legal/audit data"

Tell Interviewer:
"This requires strong consistency. Caching introduces
eventual consistency which is unacceptable here. We'll
query database directly for every request."
```

Top 10 HLD Questions: Where to Use Caching

1. Design Twitter/Instagram Feed

Cache Opportunities:

```
✅ User timelines (expensive joins)
✅ Celebrity posts (read millions of times)
✅ Trending hashtags
✅ User profiles
✅ Images/videos (CDN)
```

What to Tell Interviewer:

```
"I'd implement caching at multiple layers:

1. CDN Layer:
   - Images/videos cached at edge (1 year TTL)
   - Reduces latency: 200ms → 20ms
   - Saves 80% bandwidth costs

2. Redis Layer (5 min TTL):
```

```
   - User timelines: Post IDs + metadata
   - Profile information
   - Follower/following counts

3. Invalidation:
   - New post: Invalidate follower timelines
   - Fanout-on-write: Regular users (<1K followers)
   - Fanout-on-read: Celebrities (millions of followers)

Expected Results:
- Cache hit rate: 85-90%
- Timeline load: 50ms → 5ms
- DB load reduction: 90%
- Can serve 10M users with 10x less infrastructure
```

2. Design YouTube/Netflix

Cache Opportunities:

```
✅ Video metadata (title, views, likes)
✅ Recommendations
✅ User watch history
✅ Video content (CDN)
✅ Thumbnails
```

What to Tell Interviewer:

```
"Caching is fundamental to video streaming:

1. Content Delivery (CDN):
   - 90% of video served from edge locations
   - User in India: Served from Mumbai edge (20ms)
   - Without CDN: From US origin (200ms)
   - Cost: ~80% reduction in bandwidth

2. Metadata (Redis, 1 hour TTL):
   - Video details rarely change
   - Single video: millions of metadata requests
   - Database: 1K QPS → Cache: 100K QPS

3. Recommendations (30 min TTL):
   - Expensive ML computations (200ms)
   - Pre-compute and cache
   - Personalized per user segment

4. Performance Impact:
   - Page load: 2s → 500ms (4x faster)
   - DB queries: 10 per page → 1 per page
```

3. Design URL Shortener

Cache Opportunities:

```
✅ Popular URLs (80/20 applies heavily)
✅ URL → Original mapping
✅ Click statistics
```

What to Tell Interviewer:

```
"URL shortener is ideal for caching:

1. Extreme Read-Heavy: 100:1 ratio
   - URLs written once, read thousands of times
   - Viral links: millions of clicks
   - Top 20% URLs: 80% of all traffic

2. Simple Data Model:
   - Key-value: short → original
   - Perfect for Redis
   - Sub-millisecond lookups

3. Strategy:
   - Cache popular URLs (LRU eviction)
   - No TTL (URLs immutable)
   - Write-around (avoid cache pollution)
   - 1GB cache = 10M URLs

4. Performance:
   - Without cache: 50ms (DB query)
   - With cache: 1ms (Redis)
   - 50x improvement!
   - Cache hit rate: 95%+

This allows serving 1M QPS with minimal infrastructure."
```

4. Design Amazon E-commerce

Cache Opportunities:

```
✅ Product catalog
✅ Product images
✅
```