Interview questions - **Caching for Speed Optimization**

✅ Core Conceptual Questions

🔹 What is caching and why is it important in system design?

Answer:
Caching is the technique of storing frequently accessed data in a faster storage layer (e.g., in-memory) to reduce data retrieval time. It helps:

- Minimize latency
- Reduce backend/database load
- Improve system scalability and user experience Caching is critical in high-traffic systems where performance and responsiveness are essential.

🔹 Explain different types of caching and where they are used.

Answer:

- **Client-side cache**: Browser cache, service workers — used for static assets, localStorage.
- **Server-side cache**: In-memory (e.g., Redis, Memcached) — used in API responses or session storage.
- **CDN cache**: Distributed edge servers cache static content like images, JS, CSS.
- **Database cache**: Materialized views or query result caching — reduces repeated complex DB queries.

🔹 What are write-through vs. write-back caching strategies?

Answer:

- **Write-through**: Data is written to cache and database simultaneously. Ensures consistency but slower write performance.
- **Write-back (write-behind)**: Data is written to cache first; the DB is updated later asynchronously. Faster writes but at risk of data loss on cache failure.

🔹 What is lazy loading (cache-aside) and when would you use it?

Answer: Lazy loading (cache-aside) loads data into the cache only when needed:

- On a cache miss, fetch from DB → populate cache → return data.
- Used when not all data is frequently accessed or when cache storage is limited.
- Gives fine-grained control but requires cache invalidation management.

🧠 Scenario-Based Questions

🔹 How would you use caching to optimize a product details page?

Answer:

- Cache frequently viewed product data in Redis with a TTL.
- Use lazy loading so only requested products are cached.
- Use CDN caching for images.
- Invalidate or refresh cache on product updates. This reduces DB hits and improves page load speed significantly.

🔹 What eviction strategy would you choose for a memory-limited system?

Answer:

- **LRU (Least Recently Used)** is ideal when recently accessed items are more likely to be used again.
- **LFU (Least Frequently Used)** if certain items are accessed more often than others.
- Eviction strategy should match access patterns to avoid cache misses.

🔹 How would you keep cache and database in sync?

Answer:

- Use **write-through** for strong consistency.
- Use **cache invalidation** on data updates (manually or via message queues).
- Set appropriate TTLs to auto-refresh stale data.
- Optionally, use change data capture (CDC) mechanisms or event-driven updates.

🔹 What are the potential downsides or risks of aggressive caching?

Answer:

- **Stale data**: Cached values may be outdated if not invalidated properly.
- **Cache stampede**: Multiple requests for uncached data can hit the backend simultaneously.
- **Overuse of memory**: Poor eviction strategy can lead to inefficient memory usage.
- **Complexity**: Cache invalidation and consistency handling can increase system complexity.

🔧 Practical Implementation

🔹 How would you implement Redis caching in a web application?

Answer:

- Use Redis as a key-value store (e.g., `product:123 → productData`).
- On cache miss, fetch from DB, store in Redis with TTL.
- On cache hit, return directly from Redis.
- Use libraries like StackExchange.Redis (C#), `redis-py` (Python), or `ioredis` (Node.js).

🔹 How can you prevent cache stampedes or thundering herd problems?

Answer:

- Use **lock or mutex** during cache miss to ensure only one backend call populates the cache.
- Use **stale-while-revalidate** strategies to serve old data while refreshing in the background.
- Implement **randomized TTLs** to spread out cache expiry.

🔹 What tools can be used for distributed caching?

Answer:

- **Redis Cluster**: High availability and partitioning.
- **Memcached**: Lightweight in-memory store.
- **Hazelcast, Apache Ignite**: In-JVM distributed cache for enterprise use.
- Use consistent hashing and replication to manage cache distribution across nodes.