Contents

Performance & Scalability

This document covers performance optimization and scalability concepts for system design.

Components

Caching

- **Types of Caching**
  - **Client-side:** Browser cache, mobile app cache - reduces server load and improves user experience
  - **DNS:** Cache DNS lookups to reduce resolution time and improve initial connection speed
  - **CDN (Content Delivery Network):** Geographic distribution for static content, reduces latency through edge locations
  - **Server-side:** Web server caches (Nginx, Apache) for static/dynamic content, reduces backend processing load
  - **Application-level:** In-memory cache (Redis, Memcached) for fast access to frequently used data
  - **Database-level:** Query result caching to reduce database query execution time
  - **Disk:** Operating system and database disk caches to improve I/O performance
- **Cache Invalidation Methods**
  - **Time-based (TTL):** Expire cache entries after a set time period
  - **Event-based:** Invalidate when underlying data changes
  - **Manual/Explicit:** Purge specific objects/URLs or clear cache by application logic
  - **Pattern-based (Ban):** Invalidate cached content based on criteria (URL patterns, headers)
  - **Refresh:** Fetch latest content from origin server, update cached version
  - **Dependency-based:** Invalidate when related data changes
  - **Stale-while-revalidate:** Serve stale content immediately while updating in background
- **Cache Invalidation Schemes**
  - **Cache-aside (Lazy Loading):** Application manages cache, loads on cache miss
  - **Write-through:** Write to cache and database simultaneously
  - **Write-around:** Write directly to database, bypass cache (good for write-heavy workloads)

  - **Write-behind (Write-back):** Write to cache immediately, database asynchronously
  - **Refresh-ahead:** Proactively refresh cache before expiration
- **Cache Read Strategies**
  - **Read-through Cache:** Cache retrieves data from data store on cache miss, updates itself, and returns data to application. Cache handles all data retrieval logic.
  - **Read-aside (Cache-aside/Lazy Loading):** Application checks cache first; on cache miss, application retrieves from data store, updates cache, then uses data. Application controls caching logic.
  - **Cache-first:** Check cache first, fallback to database on miss (general pattern)
  - **Database-first:** Always read from database, update cache (for critical consistency)
  - **Cache-only:** Only read from cache (for non-critical data)
- **Cache Eviction Policies**
  - **FIFO (First In, First Out):** Evicts the first block accessed without regard to access frequency
  - **LIFO (Last In, First Out):** Evicts the most recently accessed block without regard to access frequency
  - **LRU (Least Recently Used):** Discards the least recently used items first
  - **MRU (Most Recently Used):** Discards the most recently used items first (opposite of LRU)
  - **LFU (Least Frequently Used):** Counts access frequency, discards least frequently used items
  - **Random Replacement (RR):** Randomly selects and discards items when space is needed

Performance vs. Scalability Fundamentals

- **Performance Optimization**
  - **Focus:** Speed and efficiency for current workload and user base
  - **Metrics:** Latency (response time), throughput (operations per second), resource utilization
  - **Techniques:** Code optimization, faster hardware, better algorithms, caching, indexing, connection pooling
  - **Goal:** Make the system faster under existing conditions
  - **Examples:** Optimizing database queries, using faster storage, improving algorithm complexity
  - **Time Horizon:** Immediate improvements for current scale
- **Scalability Planning**
  - **Focus:** System's ability to handle increased load (users, data, requests) over time
  - **Metrics:** Concurrent users supported, data volume capacity, geographic distribution capability
  - **Techniques:** Horizontal scaling, load balancing, data partitioning, microservices, distributed architecture
  - **Goal:** Ensure system can grow with demand without performance degradation
  - **Examples:** Adding more servers, database sharding, implementing CDNs, service decomposition
  - **Time Horizon:** Long-term capacity planning for future growth
- **Key Distinctions**
  - **Performance:** "How fast can we serve 1,000 users?" → Optimize current system

  - **Scalability:** “How do we serve 1,000,000 users?” → Redesign system architecture
  - **Performance:** Usually involves vertical improvements (better resources)
  - **Scalability:** Usually involves horizontal expansion (more resources)
  - **Performance:** May conflict with scalability (complex optimizations can reduce flexibility)
  - **Scalability:** May initially reduce performance (distributed systems have coordination overhead)

Related Trade-offs

Performance vs. Scalability

- **Summary:** Performance focuses on speed and efficiency for current loads; scalability ensures the system can grow with demand.
- **Trade-off:** Fast execution under limited load vs. system capacity for future growth.
- **Questions to Ask:**
  - What are current performance requirements?
  - What’s the expected growth in users or data volume?
  - What are the most performance-sensitive parts?
  - Do we need to scale horizontally or vertically?
  - Are we optimizing for time-to-market or long-term robustness?
  - What is our tolerance for cost in achieving performance?

Latency vs. Throughput

- **Summary:** Latency is about minimizing delay for individual operations (important for real-time systems). Throughput is about maximizing the number of operations over time (important for batch/high-volume workloads).
- **Trade-off:** Responsiveness per operation vs. total processing capacity.
- **Optimization Strategies:**
  - **Improve Latency:** CDNs for geographic proximity, in-memory caching, faster hardware/protocols, database indexing, load balancing, code optimization, minimize external calls
  - **Improve Throughput:** Horizontal scaling, parallel/batch processing, asynchronous operations, database sharding, increased network bandwidth
- **Questions to Ask:**
  - Is this a user-facing or backend system?
  - What is the maximum acceptable latency per request?
  - What’s the expected request or data volume?
  - Are we streaming, batching, or queuing data?
  - What are the business/user impacts of delay?
  - Do we need real-time guarantees?

Read-Through vs Write-Through Cache

- **Summary:** Read-through caches automatically load data from the database on cache misses, while write-through caches ensure data is written to both cache and database simultaneously. Each strategy optimizes for different access patterns and consistency requirements.

- **Trade-off:** Read performance and cache automation vs. write performance and data consistency guarantees.
- **Strategy Comparison:**
  - **Read-Through:** Cache handles data loading automatically, reduces database load for reads, but may have higher latency on cache misses
  - **Write-Through:** Ensures cache-database consistency, simplifies application logic, but slower writes due to dual write operations
  - **Cache-Aside (Alternative):** Application controls caching logic, more flexible but requires more complex application code
- **Questions to Ask:**
  - Is the workload read-heavy or write-heavy?
  - How critical is data consistency between cache and database?
  - Can the application tolerate slightly stale data?
  - What's the cache miss penalty in terms of latency?
  - Do you need immediate consistency or can you accept eventual consistency?

Server-Side Caching vs. Client-Side Caching

- **Summary:** Server-side caching stores frequently accessed data on the server to reduce database load and processing time, while client-side caching stores data locally on the user's device to improve response times and reduce network requests.
- **Trade-off:** Centralized cache control and consistency vs. reduced network load and improved user experience.
- **Caching Comparison:**
  - **Server-Side Caching:** Centralized control, shared across users, reduces backend load, but requires server resources and network round-trips for cached data
  - **Client-Side Caching:** Instant access, reduces server load, works offline, but limited storage, consistency challenges, cache invalidation complexity, security risks (sensitive data exposure), and no control over cache behavior across different clients
  - **Hybrid Approach:** Use both strategically - server-side for shared data and complex processing, client-side for user-specific data and static assets
- **Key Client-Side Caching Cons:**
  - **Data Inconsistency:** Users may see stale data when underlying data changes
  - **Security Risks:** Sensitive data stored locally can be compromised
  - **Storage Limitations:** Limited space on client devices, especially mobile
  - **Cache Invalidation Complexity:** Difficult to ensure all clients update when data changes
  - **No Centralized Control:** Cannot guarantee cache behavior across different devices/browsers
- **Questions to Ask:**
  - What type of data are you caching (user-specific vs. shared)?
  - How important is data consistency across users?
  - What are the network latency and bandwidth constraints?
  - Do users need offline access to the application?
  - How frequently does the cached data change?
  - What's the tolerance for stale data on the client?
  - Are there storage limitations on client devices?
  - Does the cached data contain sensitive information?

CDN Usage vs. Direct Server Serving

- **Summary:** Content Delivery Networks (CDNs) cache content at geographically distributed edge servers to reduce latency and server load, while direct server serving handles all requests from origin servers. Each approach has different implications for performance, cost, and complexity.
- **Trade-off:** Global performance optimization and reduced server load vs. simplicity and direct control.
- **Delivery Comparison:**
  - **CDN Usage:** Faster global content delivery, reduced origin server load, better availability and DDoS protection, but additional cost, cache invalidation complexity, and less control over content delivery
  - **Direct Server Serving:** Full control over content delivery, simpler architecture, no additional CDN costs, but higher latency for distant users, increased server load, and potential bandwidth costs
  - **Hybrid Approach:** Use CDN for static assets (images, CSS, JS) and frequently accessed content, direct serving for dynamic, personalized, or sensitive content
- **Questions to Ask:**
  - What types of content are you serving (static vs. dynamic)?
  - Do you have a global user base or regional concentration?
  - What are your latency and performance requirements?
  - How frequently does content change and need cache invalidation?
  - What's your tolerance for additional cost vs. performance gains?
  - Do you need real-time content updates or can you accept some staleness?
  - Are there compliance requirements for data locality?
  - What's your current server capacity and bandwidth costs?