System Design Terminology Guide - Essential Concepts for Scale

Table of Contents

Introduction

> **"Knowing the vocabulary is half the battle in system design interviews."**

This guide covers every important term you need to know to discuss systems at scale. For each term:

- ✅ Clear definition
- ✅ When to use it
- ✅ Real-world examples
- ✅ Interview context

Scalability Concepts

1. Scalability

Definition: A system's ability to handle increased load by adding resources

Two Types:

Vertical Scaling (Scale Up):

- Add more power to existing machine
- Bigger CPU, more RAM, faster disk
- Example: Upgrade from 8GB to 32GB RAM
- **Limit**: Single machine ceiling (~1TB RAM)
- **Pros**: Simple, no code changes
- **Cons**: Expensive, single point of failure

Horizontal Scaling (Scale Out):

- Add more machines
- Distribute load across servers
- Example: 1 server → 10 servers → 100 servers
- **Limit**: Nearly unlimited
- **Pros**: Cost-effective, fault-tolerant
- **Cons**: Complex, requires distributed architecture

Interview Usage:
"We'd start with vertical scaling for simplicity, then move to horizontal scaling when traffic exceeds 10K QPS"

---

2. Load Balancer

Definition: Distributes incoming traffic across multiple servers

Types:

Layer 4 (Transport Layer):

- Routes based on IP address and port
- Fast, simple
- No visibility into application data
- Example: TCP/UDP load balancing

Layer 7 (Application Layer):

- Routes based on HTTP headers, URLs, cookies
- Can route /api/users to user service, /api/posts to post service
- Slower but more flexible
- Example: NGINX, HAProxy, AWS ALB

Algorithms:

- **Round Robin**: Distribute sequentially (server1, server2, server3, server1...)
- **Least Connections**: Route to server with fewest active connections
- **Least Response Time**: Route to fastest responding server
- **IP Hash**: Same client always routes to same server (sticky sessions)

Real Examples:

- **Netflix**: AWS ELB distributing traffic across thousands of instances
- **Twitter**: HAProxy for internal service routing

Interview Usage:
"I'd use Layer 7 load balancer to route API requests to appropriate microservices and handle SSL termination"

---

3. CDN (Content Delivery Network)

Definition: Globally distributed network of servers that cache content close to users

How It Works:

```
User in Tokyo → Requests image
             → CDN edge server in Tokyo (cache hit)
             → Returns image in 10ms

vs. without CDN:
User in Tokyo → Origin server in California
             → Returns image in 150ms (15x slower!)
```

Types:

Push CDN:

- You upload content to CDN manually
- Good for: Infrequently changing content
- Example: Uploading product catalog

Pull CDN:

- CDN pulls content from origin when first requested
- Good for: Frequently changing content
- Example: Social media posts

Real Examples:

- **Netflix**: Uses AWS CloudFront for streaming
- **Instagram**: Akamai for photo delivery
- **Twitter**: Fastly for image caching

Interview Usage:
"For media-heavy applications like Instagram, CDN is mandatory, not optional. It reduces origin bandwidth by 90%+ and improves latency from 150ms to 10ms"

---

4. Caching

Definition: Store frequently accessed data in fast storage (RAM) to avoid slow disk/database reads

Cache Levels:

```
Browser Cache → CDN Cache → Server Cache → Database Cache
(client-side)  (edge)      (application) (built-in)
```

Cache Strategies:

Cache-Aside (Lazy Loading):

```
1. Check cache
2. If miss → Query database → Store in cache
3. If hit → Return immediately

Use for: Read-heavy workloads
Example: User profiles, product catalog
```

Write-Through:

```
1. Write to cache
2. Cache synchronously writes to database
3. Return

Use for: Strong consistency needed
Example: Financial transactions
```

Write-Behind (Write-Back):

```
1. Write to cache
2. Return immediately
3. Cache asynchronously writes to database later

Use for: High write throughput
Example: Analytics events, logs
```

Cache Eviction Policies:

- **LRU** (Least Recently Used): Remove oldest accessed item
- **LFU** (Least Frequently Used): Remove least accessed item
- **FIFO** (First In First Out): Remove oldest item
- **TTL** (Time To Live): Auto-expire after time period

Real Examples:

- **Facebook**: Memcached for caching everything (photos, profiles, etc.)
- **Twitter**: Redis for timeline caching
- **Reddit**: Uses Memcached extensively

Interview Usage:
"I'd implement cache-aside pattern with Redis. This would reduce database load by 80% by caching hot user profiles and timelines"

---

5. Database Replication

Definition: Copy data across multiple database servers

Master-Slave Replication:

```
[Master] ──writes──→ DB
    ↓
    └──replicates──→ [Slave 1] ──reads──→ Users
                  → [Slave 2] ──reads──→ Users
                  → [Slave 3] ──reads──→ Users

Pros: Scales reads (add more slaves)
Cons: Doesn't scale writes, replication lag
```

Master-Master Replication:

```
[Master 1] ←──sync──→ [Master 2]
    ↓                      ↓
  Writes                 Writes
  Reads                  Reads

Pros: Scales reads AND writes, high availability
Cons: Conflict resolution needed, complex
```

Replication Lag:

- Time for data to propagate from master to slave
- Typically milliseconds to seconds
- Can cause stale reads

Real Examples:

- **Instagram**: 12 PostgreSQL replicas in different zones
- **GitHub**: MySQL with multiple read replicas

Interview Usage:
"For read-heavy workload, I'd use master-slave replication with 5 read replicas to handle 50K read QPS"

---

6. Sharding (Horizontal Partitioning)

Definition: Split data across multiple databases, each holding a subset

Sharding Strategies:

Range-Based Sharding:

```
Users 1-1M     → Shard 1
Users 1M-2M    → Shard 2
Users 2M-3M    → Shard 3

Pros: Simple to implement
Cons: Uneven distribution (hotspots)
```

Hash-Based Sharding:

```
Shard = hash(user_id) % num_shards

User 12345 → hash(12345) % 4 = 1 → Shard 1
User 67890 → hash(67890) % 4 = 2 → Shard 2

Pros: Even distribution
Cons: Hard to add/remove shards
```

Geographic Sharding:

```
US users → US shard
EU users → EU shard
Asia users → Asia shard

Pros: Low latency (data close to users)
Cons: Uneven distribution
```

Consistent Hashing:

```
Improves on hash-based sharding
When adding/removing nodes, only K/n keys need to be remapped
(K = total keys, n = number of nodes)

Used by: Cassandra, DynamoDB, Redis Cluster
```

Trade-offs:

- ✅ Scales both reads and writes
- ❌ Can't JOIN across shards
- ❌ Complex queries become difficult
- ❌ Rebalancing is expensive

Real Examples:

- **Instagram**: Shards PostgreSQL by user_id

- **Twitter**: Shards by tweet_id
- **Uber**: Geographic sharding for low latency

Interview Usage:
"When single PostgreSQL instance can't handle 10K+ write QPS, I'd shard by user_id using consistent hashing to distribute load evenly"

---

7. Federation (Vertical Partitioning)

Definition: Split database by function, not by rows

Example:

```
Single DB:
[Database] → Users, Products, Orders, Reviews

Federated:
[Users DB] → User tables only
[Products DB] → Product tables only
[Orders DB] → Order tables only

Pros:
- Smaller databases (fit in memory)
- Better cache locality
- Parallel writes

Cons:
- Can't JOIN across databases
- Application complexity
```

When to Use:

- Before sharding (simpler)
- Clear functional boundaries
- Different access patterns per function

Real Examples:

- **Amazon**: Separate databases for users, orders, inventory
- **Netflix**: Separate DBs for user data, viewing history, recommendations

Interview Usage:
"As first scaling step, I'd federate into Users DB and Content DB before implementing sharding"

Reliability & Availability

8. High Availability (HA)

Definition: System remains operational despite failures

Measured in "Nines":

```
99%       (2 nines) = 3.65 days downtime/year
99.9%     (3 nines) = 8.76 hours downtime/year
99.99%    (4 nines) = 52.6 minutes downtime/year
99.999%   (5 nines) = 5.26 minutes downtime/year
99.9999%  (6 nines) = 31.5 seconds downtime/year

Most companies target: 99.9% - 99.99%
```

Achieving HA:

- **Redundancy**: Multiple copies of everything
- **Failover**: Automatic switch to backup
- **Load Balancing**: Distribute across multiple servers
- **Health Checks**: Detect and route around failures
- **Circuit Breaker**: Fail fast instead of hanging

Real Examples:

- **AWS**: 99.99% SLA for most services
- **Google**: 99.95% SLA for Search
- **Netflix**: Designed for failure (Chaos Monkey)

Interview Usage:
"To achieve 99.99% availability (4 nines), I'd deploy across 3 availability zones with automatic failover"

---

9. Fault Tolerance

Definition: System continues operating even when components fail

Techniques:

Redundancy:

- N+1: One extra backup
- N+2: Two extra backups
- Example: 10 servers needed + 2 backup = 12 servers

Replication:

- Data copied to multiple nodes
- Typically 3 replicas
- Example: Cassandra writes to 3 nodes

Graceful Degradation:

- Core features work even if some components fail
- Example: Twitter timeline works even if trending topics fail

Real Examples:

- **Netflix**: If recommendation engine fails, show popular content
- **Amazon**: If product reviews fail, still show product page

Interview Usage:
"For fault tolerance, I'd replicate data across 3 nodes and implement graceful degradation where search failure doesn't break core features"

---

10. CAP Theorem

Definition: In distributed system, you can only have 2 of 3:

C - Consistency: All nodes see same data simultaneously
A - Availability: Every request gets a response (success or failure)
P - Partition Tolerance: System works despite network failures

Since network partitions WILL happen, you must choose:

CP (Consistency + Partition Tolerance):

- Sacrifice availability during partition
- Example: **MongoDB**, PostgreSQL, Redis
- Use for: Banking, inventory (can't show wrong balance)

AP (Availability + Partition Tolerance):

- Sacrifice consistency during partition
- Example: **Cassandra**, DynamoDB, Couchbase
- Use for: Social feeds, analytics (stale data acceptable)

Real Examples:

- **Bank Account**: CP (must show correct balance)
- **Twitter Feed**: AP (stale tweets acceptable)
- **Shopping Cart**: AP (eventual consistency OK)

Interview Usage:
"For user authentication, I'd choose CP (strong consistency) because showing wrong account balance is unacceptable. For the social feed, AP is fine since eventual consistency is acceptable"

---

11. Consistency Models

Definition: How quickly changes propagate in distributed system

Strong Consistency:

```
Write completes → All subsequent reads see the write
Example: Bank transactions

Timeline:
T1: Write(x = 5)
T2: Read() → Returns 5 (guaranteed)
```

Eventual Consistency:

```
Write completes → Reads EVENTUALLY see the write (may take time)
Example: DNS updates, social media feeds

Timeline:
T1: Write(x = 5)
T2: Read() → Might return old value (4)
T3: Read() → Returns 5 (eventually)
```

Causal Consistency:

```
If operation A causes operation B, everyone sees A before B
Example: Reply always appears after original post

Timeline:
T1: Post created
T2: Reply to post
T3: All users see post before reply (causality preserved)
```

Real Examples:

- **Strong**: PostgreSQL, MySQL, DynamoDB (optional)
- **Eventual**: Cassandra, DynamoDB (default), DNS
- **Causal**: Some distributed databases

Interview Usage:
"User profiles need strong consistency, but for the news feed, eventual consistency is acceptable since users won't notice if a post appears 1-2 seconds later"

Performance Concepts

12. Latency

Definition: Time taken to complete an operation

P50, P95, P99 Percentiles:

```
P50 (Median): 50% of requests faster than this
P95: 95% of requests faster than this
P99: 99% of requests faster than this

Example:
If P99 = 100ms, then 99% of requests complete in < 100ms
1% of requests take > 100ms (tail latency)

Why P99 matters:
- User with 100 parallel requests
- Probability all succeed in P50 time: 0.5^100 ≈ 0% (practically 0)
- At least one hits P99: Very high probability
```

Target Latencies:

```
Cache read:          < 1ms
Database read:       < 10ms
API response:        < 100ms
Page load:           < 1 second
Video start:         < 2 seconds
```

Interview Usage:
"Target p99 latency of 100ms for API responses, which means 99% of users get response in < 100ms"

---

13. Throughput

Definition: Number of operations per unit time (usually QPS)

vs. Latency:

- **Latency**: How long ONE request takes
- **Throughput**: How many requests in given time

Example:

```
System A: 10ms latency, 1,000 QPS throughput
System B: 20ms latency, 10,000 QPS throughput

System B is slower per request but handles more requests!
```

Interview Usage:
"System needs to handle 100K QPS throughput with p99 latency under 200ms"

---

14. Bottleneck

Definition: Component that limits overall system performance

Common Bottlenecks:

- **Database**: Too many queries, slow disk I/O
- **Network**: Bandwidth limit reached
- **CPU**: Complex computations, encryption
- **Memory**: Cache full, OOM errors
- **Disk I/O**: Database writes, log writes

How to Identify:

- Monitor CPU, memory, disk, network
- Find component at 80-100% utilization
- That's your bottleneck!

Interview Usage:
"At 10K write QPS, the database becomes the bottleneck. I'd add write-optimized database like Cassandra"

Database Concepts

15. ACID

Definition: Properties of reliable database transactions

A - Atomicity: Transaction is all-or-nothing

```
Transfer $100:
1. Deduct $100 from Account A
2. Add $100 to Account B

If step 2 fails, step 1 rolls back
Either both happen or neither happens
```

C - Consistency: Database remains in valid state

```
If Account A – $100, then total money in system unchanged
Constraints always satisfied
```

I - Isolation: Concurrent transactions don't interfere

```
Transaction 1: Read balance, transfer money
Transaction 2: Read balance, transfer money
```

```
Results same as if run sequentially
```

D - Durability: Committed data persists

```
After transaction commits, data survives crashes
Written to disk, not just memory
```

Databases with ACID:

- PostgreSQL, MySQL, Oracle, SQL Server

Interview Usage:
"For payment processing, I need ACID guarantees, so I'd use PostgreSQL"

---

16. BASE

Definition: Properties of NoSQL databases (alternative to ACID)

BA - Basically Available: System always responds (might be stale)
S - Soft State: State may change without input (replication happening)
E - Eventual Consistency: Eventually all nodes will be consistent

Example:

```
You update your Twitter bio:
- Write succeeds immediately (Basically Available)
- Some replicas still show old bio (Soft State)
- After 1 second, all replicas show new bio (Eventual Consistency)
```

Databases with BASE:

- Cassandra, DynamoDB, MongoDB (default), Couchbase

Interview Usage:
"For social media feed, BASE model is acceptable since users can tolerate seeing slightly stale data"

---

17. Normalization vs Denormalization

Normalization: Organize data to reduce redundancy

```
Users Table:
user_id | name
1       | Alice
```

```
2       | Bob

Orders Table:
order_id | user_id | product
100      | 1       | Laptop
101      | 1       | Mouse

No data duplication
Need JOIN to get user name with order
```

Denormalization: Duplicate data to avoid JOINs

```
Orders Table:
order_id | user_id | user_name | product
100      | 1       | Alice     | Laptop
101      | 1       | Alice     | Mouse

Data duplicated (Alice appears twice)
No JOIN needed, faster reads
```

Trade-offs:

- **Normalized**: Slower reads (JOINs), easier writes, no redundancy
- **Denormalized**: Faster reads, complex writes (update many places), redundancy

Real Examples:

- **Facebook**: Denormalizes for timeline performance
- **Twitter**: Denormalizes user data in tweets

Interview Usage:
"I'd denormalize by storing user name with each tweet to avoid JOINs when loading timeline, trading write complexity for read speed"

---

18. Indexing

Definition: Data structure that improves query speed

How It Works:

```
Without Index:
SELECT * FROM users WHERE email = 'alice@example.com'
→ Scans entire table (O(n))
→ 1 billion users = very slow!

With Index on email:
→ B-tree lookup (O(log n))
```

```
→ 1 billion users = ~30 comparisons
→ Sub-millisecond query
```

Types:

- **Primary Index**: On primary key (automatic)
- **Secondary Index**: On other columns
- **Composite Index**: On multiple columns
- **Unique Index**: Enforces uniqueness

Trade-offs:

- ✅ Faster reads (10-1000x)
- ❌ Slower writes (must update index)
- ❌ More storage (index size)

Interview Usage:
"I'd create index on (user_id, created_at) for efficient timeline queries sorted by time"

Distributed Systems

19. Distributed System

Definition: Multiple computers working together as single system

Challenges:

- **Network is unreliable**: Packets drop, partitions happen
- **Latency**: Cross-datacenter calls take 50-150ms
- **No global clock**: Can't know exact order of events
- **Partial failures**: Some nodes work, some don't

Benefits:

- **Scalability**: Add more machines
- **Fault Tolerance**: One machine fails, others continue
- **Geographic Distribution**: Serve users globally

Real Examples:

- **Google**: Thousands of datacenters worldwide
- **Netflix**: Distributed across AWS regions
- **Facebook**: Multiple datacenters, cross-datacenter replication

Interview Usage:
"In distributed system, we must handle network partitions, so per CAP theorem, I'd choose between consistency and availability"

---

20. Consensus

Definition: Multiple nodes agreeing on a single value

Why Needed:

- Leader election (who's the master?)
- Configuration management (what's the current state?)
- Distributed locks (who has access?)

Algorithms:

- **Paxos**: Complex but proven
- **Raft**: Simpler, easier to understand
- **Zab**: Used by Zookeeper

Technologies:

- **Zookeeper**: Apache's consensus service
- **etcd**: Kubernetes uses this
- **Consul**: HashiCorp's service discovery

Interview Usage:
"For distributed counter in URL shortener, I'd use Zookeeper for consensus to ensure unique IDs"

---

21. Message Queue

Definition: Asynchronous communication between services

How It Works:

```
[Producer] → [Message Queue] → [Consumer]
               (Kafka)

Producer: Sends messages without waiting
Queue: Stores messages durably
Consumer: Processes messages at own pace
```

Use Cases:

- **Decouple services**: Producer doesn't depend on consumer
- **Handle spikes**: Queue buffers during traffic surge
- **Async processing**: Don't block user request
- **Retry logic**: Reprocess failed messages

Popular Queues:

- **Kafka**: High throughput, replay capability
- **RabbitMQ**: Flexible routing, AMQP protocol

- **Amazon SQS**: Managed, simple
- **Redis**: Simple, fast (but can lose messages)

Real Examples:

- **Uber**: Kafka for trip events, surge pricing
- **LinkedIn**: Kafka for activity streams
- **Instagram**: RabbitMQ for feed fanout

Interview Usage:
"I'd use Kafka message queue for asynchronous feed fanout, so posting a tweet doesn't block while updating millions of followers' timelines"

---

22. Microservices

Definition: Application as collection of small, independent services

vs. Monolith:

```
Monolith:
[Single Application]
├─ User Module
├─ Post Module
├─ Payment Module
└─ Notification Module

Microservices:
[User Service] ←→ [Post Service]
      ↓                ↓
[Payment Service] ←→ [Notification Service]
```

Benefits:

- **Independent scaling**: Scale hot services
- **Independent deployment**: Deploy without affecting others
- **Technology flexibility**: Each service can use different tech
- **Team autonomy**: Teams own services

Challenges:

- **Network overhead**: HTTP calls between services
- **Distributed debugging**: Trace requests across services
- **Data consistency**: Each service has own database
- **Operational complexity**: Deploy/monitor many services

Real Examples:

- **Netflix**: 1000+ microservices

- **Uber**: 2000+ microservices
- **Amazon**: Service-oriented architecture

Interview Usage:
"I'd use microservices architecture with User Service, Post Service, and Feed Service, each scaling independently"

---

23. API Gateway

Definition: Single entry point for all client requests

Responsibilities:

- **Authentication**: Validate JWT tokens
- **Rate Limiting**: Prevent abuse
- **Request Routing**: Route to appropriate microservice
- **Response Aggregation**: Combine multiple service calls
- **Protocol Translation**: REST to gRPC
- **SSL Termination**: Handle HTTPS

Technologies:

- Kong, AWS API Gateway, Apigee, Express Gateway

Interview Usage:
"API Gateway handles authentication and rate limiting before routing requests to microservices"

Load Balancing

24. Sticky Sessions (Session Affinity)

Definition: Route same user to same server

How It Works:

```
User Alice → Server 1 (first request)
User Alice → Server 1 (all subsequent requests)
User Bob → Server 2 (first request)
User Bob → Server 2 (all subsequent requests)
```

When Needed:

- Server stores session in memory
- WebSocket connections
- File uploads (maintain state)

Problems:

- Uneven distribution (hot users)
- If server dies, lose session
- Better: Store session in Redis (shared)

Interview Usage:
"I'd avoid sticky sessions by storing sessions in Redis, making servers stateless for easier scaling"

---

25. Health Checks

Definition: Automatically detect and remove unhealthy servers

How It Works:

```
Load Balancer sends request every 10 seconds:
GET /health

If server responds 200 OK → Healthy
If timeout or error → Unhealthy → Remove from pool
```

Health Endpoint:

```python
@app.route('/health')
def health_check():
    # Check database connection
    if not db.ping():
        return "Unhealthy", 503

    # Check cache connection
    if not redis.ping():
        return "Unhealthy", 503

    return "Healthy", 200
```

Interview Usage:
"Load balancer performs health checks every 10 seconds, automatically removing unhealthy instances"

Caching Strategies

26. Cache Invalidation

Definition: Removing stale data from cache

Problem: "There are only two hard things in Computer Science: cache invalidation and naming things" - Phil Karlton

Strategies:

TTL (Time To Live):

```
SET user:123 "data" EX 3600  # Expire in 1 hour
Pros: Simple, automatic
Cons: Data might be stale before expiry
```

Write-Through Invalidation:

```
When data updated:
1. Update database
2. Update cache
Pros: Cache always fresh
Cons: Extra write to cache
```

Write-Invalidate:

```
When data updated:
1. Update database
2. Delete from cache
3. Next read will reload from DB
Pros: Simpler than write-through
Cons: Cache miss after every write
```

Interview Usage:
"I'd use TTL-based expiration for user profiles (1 hour TTL) and active invalidation for critical updates like password changes"

---

27. Cache Stampede (Thundering Herd)

Definition: Many requests hit database simultaneously when cache expires

Problem:

```
Popular cache entry expires
1000 requests arrive simultaneously
All 1000 miss cache
All 1000 query database → Database overwhelmed!
```

Solution:

```
def get_with_lock(key):
    value = cache.get(key)
```

```
    if value:
        return value

    # Acquire lock
    if cache.set_nx(f"lock:{key}", "1", ex=10):
        # Only one request queries DB
        value = db.query(key)
        cache.set(key, value, ex=3600)
        cache.delete(f"lock:{key}")
        return value
    else:
        # Others wait and retry
        time.sleep(0.1)
        return get_with_lock(key)  # Retry
```

Interview Usage:
"To prevent cache stampede on popular items, I'd use distributed locks so only one request regenerates cache"

Data Partitioning

28. Consistent Hashing

Definition: Hash function that minimizes key redistribution when nodes added/removed

Problem with Simple Hashing:

```
hash(key) % N

Add/remove server → All keys need to be remapped!
Example: 4 servers → 5 servers
Only 1/5 of keys map to same server
4/5 need to move = 80% of data!
```

Consistent Hashing Solution:

```
Place servers on hash ring
Place keys on hash ring
Key goes to next server clockwise

Add/remove server → Only K/N keys remapped
(K = total keys, N = servers)

Example: 4 servers → 5 servers
Only 1/5 of keys move = 20% of data
```

Virtual Nodes:

```
Each physical server has multiple positions on ring
Better distribution, handles heterogeneous servers

Large server: 1000 virtual nodes
Small server: 100 virtual nodes
```

Used By:

- **Cassandra**: Data distribution
- **DynamoDB**: Partitioning
- **Redis Cluster**: Sharding
- **CDNs**: Server selection

Interview Usage:
"I'd use consistent hashing for cache distribution so adding cache servers doesn't cause massive key remapping"

---

29. Hot Partition (Hotspot)

Definition: One partition/shard receiving disproportionate traffic

Problem:

```
Celebrity user with 10M followers
All their data in one shard
Shard overloaded while others idle

Example:
Shard 1 (celebrity): 50K QPS
Shard 2 (normal users): 1K QPS
Shard 3 (normal users): 1K QPS
```

Solutions:

- **Partition by content, not user**: Tweet ID instead of user ID
- **Replicate hot data**: Multiple copies of celebrity data
- **Cache aggressively**: Serve hot data from cache

Real Examples:

- **Twitter**: Handles celebrity tweets differently (fan-out on read)
- **Instagram**: Caches celebrity profiles heavily

Interview Usage:
"To avoid hot partition with celebrity users, I'd shard by tweet_id not user_id, and cache celebrity data

aggressively"

Architecture Patterns

30. CQRS (Command Query Responsibility Segregation)

Definition: Separate read and write data models

Traditional:

```
[Application] ←→ [Single Database]
  (reads & writes)
```

CQRS:

```
[Application]
    ├──writes──→ [Write DB] (PostgreSQL)
    │                ↓
    │           [Sync/Event]
    │                ↓
    └──reads──→ [Read DB] (Cassandra, cache-optimized)
```

Benefits:

- **Optimize separately**: Write DB for writes, Read DB for reads
- **Independent scaling**: Scale reads without affecting writes
- **Different models**: Denormalized for reads, normalized for writes

Use Cases:

- Social media (write tweets to Cassandra, read from Redis cache)
- E-commerce (write orders to PostgreSQL, read from MongoDB)
- Analytics (write events to Kafka, read from Redshift)

Real Examples:

- **Netflix**: Writes to Cassandra, reads from various caches
- **Amazon**: Order writes to Oracle, inventory reads from cache

Interview Usage:
"I'd implement CQRS with writes to PostgreSQL and reads from Redis cache for optimal performance"

---

31. Event Sourcing

Definition: Store all changes as sequence of events, not final state

Traditional (State-based):

```
User Account:
balance = $100

After deposit $50:
balance = $150 (old value lost)
```

Event Sourcing:

```
Event Log:
1. Account created, balance = $100
2. Deposited $50, balance = $150
3. Withdrew $20, balance = $130

Current state = Replay all events
Can rebuild state at any point in time
```

Benefits:

- **Audit trail**: Complete history
- **Time travel**: See state at any point
- **Debugging**: Replay events to reproduce bugs
- **Event replay**: Reprocess events with new logic

Combined with CQRS:

```
Commands → Event Store (Kafka) → Read Models
                                  (Multiple DBs)
```

Real Examples:

- **Banking**: Transaction history as events
- **E-commerce**: Order state changes as events
- **Git**: Commits are events!

Interview Usage:
"For audit requirements, I'd use event sourcing where all order state changes are stored as events in Kafka"

---

32. Circuit Breaker

Definition: Prevent cascading failures by failing fast

States:

```
CLOSED (Normal):
  Requests pass through
  If failures exceed threshold → OPEN

OPEN (Failing):
  Reject requests immediately (fail fast)
  After timeout → HALF_OPEN

HALF_OPEN (Testing):
  Allow few requests through
  If succeed → CLOSED
  If fail → OPEN
```

Example:

```python
class CircuitBreaker:
    def __init__(self, failure_threshold=5, timeout=60):
        self.failures = 0
        self.threshold = failure_threshold
        self.state = "CLOSED"
        self.timeout = timeout

    def call(self, func):
        if self.state == "OPEN":
            raise Exception("Circuit breaker OPEN")

        try:
            result = func()
            self.failures = 0  # Reset on success
            return result
        except Exception:
            self.failures += 1
            if self.failures >= self.threshold:
                self.state = "OPEN"
                schedule_timeout(self.timeout)
            raise
```

Benefits:

- Prevent cascading failures
- Give failing service time to recover
- Better error messages

Real Examples:

- **Netflix Hystrix**: Circuit breaker library

- **Uber**: Protects against downstream service failures

Interview Usage:
"I'd implement circuit breaker pattern so if recommendation service fails, we fail fast instead of timing out"

---

33. Rate Limiting

Definition: Limit number of requests per time period

Algorithms:

Token Bucket:

```
Bucket holds N tokens
Every request consumes 1 token
Refill R tokens per second
If no tokens → Reject request

Example:
Capacity: 100 tokens
Refill: 10 tokens/second
Allows burst of 100, then 10/second sustained
```

Leaky Bucket:

```
Requests enter bucket
Process at fixed rate
If bucket full → Reject

Example:
Process 10 requests/second, no burst allowed
```

Fixed Window:

```
100 requests per hour
Count resets at hour boundary

Problem: 100 at 12:59, 100 at 13:00 = 200 in 1 minute!
```

Sliding Window:

```
100 requests per rolling 60-minute window
Counts requests in last hour from current time
```

Real Examples:

- **Twitter API**: 300 requests per 15 minutes
- **GitHub API**: 5000 requests per hour
- **Stripe**: Prevents payment spam

Interview Usage:
"I'd implement token bucket algorithm in Redis for rate limiting - 100 requests per hour per user"

---

34. Idempotency

Definition: Operation can be applied multiple times with same result

Idempotent:

```
SET balance = 100
Call once: balance = 100
Call twice: balance = 100 (same result)

DELETE user_id = 5
Call once: User deleted
Call twice: User already deleted (same result, no error)
```

Not Idempotent:

```
INCREASE balance by 10
Call once: balance = 110
Call twice: balance = 120 (different result!)
```

Why Important:

- Network failures require retries
- Can safely retry idempotent operations
- Non-idempotent needs deduplication

Making Operations Idempotent:

```python
# Use unique request ID
def process_payment(payment_id, amount):
    if redis.exists(f"processed:{payment_id}"):
        return "Already processed"

    # Process payment
    charge_card(amount)
```

```
    # Mark as processed
    redis.set(f"processed:{payment_id}", "1", ex=86400)
    return "Success"
```

Interview Usage:
"For payment API, I'd make it idempotent using request IDs so retries don't charge customer twice"

---

35. Bloom Filter

Definition: Space-efficient probabilistic data structure to test set membership

How It Works:

```
Use k hash functions
Set k bits in bit array

Check membership:
- If all k bits are 1 → Probably in set
- If any bit is 0 → Definitely not in set

False positive: Possible (says yes when actually no)
False negative: Impossible (never says no when actually yes)
```

Use Cases:

- **Check if username exists** (before DB query)
- **Check if URL is malicious** (before fetching)
- **Weak password detection** (check against common passwords)
- **Cache key existence** (avoid cache misses)

Space Savings:

```
1 billion URLs
100 bytes each = 100 GB

Bloom filter (1% false positive):
1 billion URLs = ~1.2 GB (83x smaller!)
```

Real Examples:

- **Google Chrome**: Check malicious URLs
- **Medium**: Recommend articles you haven't read
- **Bitcoin**: Check if transaction exists

Interview Usage:
"I'd use Bloom filter to quickly check if shortened URL exists before querying database, saving 99% of queries for non-existent URLs"

---

36. Geohashing

Definition: Encode geographic coordinates into short string

How It Works:

```
Latitude, Longitude → Short string
(37.7749, -122.4194) → "9q8yy" (San Francisco)

Nearby locations have similar prefixes:
9q8yy1  ← Close
9q8yy2  ← Close
9q9xyz  ← Far
```

Use Cases:

- **Uber**: Find nearby drivers
- **Yelp**: Find nearby restaurants
- **Pokemon Go**: Spawn nearby Pokemon
- **Tinder**: Find matches in area

Benefits:

- Simple range query for nearby items
- Works with standard databases
- Variable precision (longer = more precise)

Interview Usage:
"For Uber driver matching, I'd use geohashing to efficiently query drivers within 5km radius using prefix matching"

---

37. WebSocket

Definition: Persistent bidirectional connection between client and server

vs. HTTP:

```
HTTP (Request-Response):
Client → Request → Server
Client ← Response ← Server
(Connection closes)

WebSocket (Persistent):
```

```
Client ←──────→ Server
(Connection stays open, messages flow both ways)
```

Use Cases:

- **Chat applications**: Real-time messaging
- **Live updates**: Stock prices, sports scores
- **Collaborative editing**: Google Docs
- **Gaming**: Multiplayer games
- **Notifications**: Push notifications

Benefits:

- Low latency (no connection overhead)
- Server can push to client
- Efficient (one connection, many messages)

Challenges:

- Maintain many connections (memory)
- Load balancing with sticky sessions
- Scaling WebSocket servers

Real Examples:

- **Slack**: Chat messages
- **WhatsApp Web**: Message sync
- **Trading platforms**: Live prices

Interview Usage:
"For real-time chat, I'd use WebSocket for bidirectional communication instead of polling, reducing load by 100x"

---

38. Long Polling

Definition: Client requests, server holds connection until data available

How It Works:

```
1. Client sends request
2. Server holds connection open (30–60 seconds)
3. When data available OR timeout:
   – Server responds
   – Client immediately sends new request
```

vs. Short Polling:

```
Short Polling (wasteful):
Check every second → 3600 requests/hour
99% are "no new data"

Long Polling (efficient):
Hold connection → 60 requests/hour
Only responds when data available
```

Use Cases:

- Chat applications (before WebSocket)
- Notification systems
- Real-time updates

Interview Usage:
"Before WebSocket, I'd use long polling to reduce polling overhead from 3600 requests/hour to 60 requests/hour"

---

39. Server-Sent Events (SSE)

Definition: Server pushes updates to client over HTTP

vs. WebSocket:

```
WebSocket: Bidirectional
SSE: Unidirectional (server → client only)

SSE is simpler when only server needs to push
```

Use Cases:

- Live sports scores
- Stock price updates
- News feed updates
- Server status monitoring

Interview Usage:
"For live stock prices where client only receives updates, I'd use SSE instead of WebSocket for simplicity"

---

40. Reverse Proxy

Definition: Server that sits in front of web servers and forwards requests

vs. Forward Proxy:

```
Forward Proxy (client-side):
Client → Proxy → Internet
Hides client identity

Reverse Proxy (server-side):
Client → Proxy → Backend Servers
Hides server identity
```

Benefits:

- **Load balancing**: Distribute requests
- **SSL termination**: Handle HTTPS
- **Caching**: Cache responses
- **Compression**: Gzip responses
- **Security**: Hide backend servers

Technologies:

- NGINX, HAProxy, Apache

Interview Usage:
"I'd use NGINX as reverse proxy for SSL termination and response caching"

---

41. Fan-out

Definition: One message triggers multiple parallel operations

Example (Twitter):

```
User with 1M followers tweets

Fan-out on Write (Push):
  Tweet posted → Write to 1M follower timelines

Fan-out on Read (Pull):
  Tweet posted → Store once
  Follower requests timeline → Fetch tweets from all following
```

Hybrid Approach:

```
Normal users (< 100K followers): Fan-out on write
Celebrities (> 100K followers): Fan-out on read
```

Real Examples:

- **Twitter**: Hybrid fan-out for timeline
- **Instagram**: Push notifications fan out to followers
- **Facebook**: News feed fan-out

Interview Usage:
"For timeline generation, I'd use hybrid fan-out: push for normal users, pull for celebrities to avoid the celebrity problem"

---

42. Back Pressure

Definition: Slow down producer when consumer can't keep up

Problem:

```
Producer: 10K messages/second
Consumer: 1K messages/second
Queue grows indefinitely → Memory exhausted
```

Solution:

```
When queue size > threshold:
  - Reject new messages (HTTP 503)
  - Client retries with exponential backoff
  - Or: Producer blocks until space available
```

Interview Usage:
"If Kafka queue backs up, I'd apply back pressure by rejecting new events and signaling clients to slow down"

---

43. Bulkhead Pattern

Definition: Isolate resources to prevent total system failure

Example:

```
Without Bulkhead:
[Shared Thread Pool - 100 threads]
  Used by: API, Background Jobs, Admin

If background jobs consume all 100 threads →
API requests blocked!

With Bulkhead:
[API Thread Pool - 70 threads]
[Background Jobs - 20 threads]
```

```
[Admin – 10 threads]

Background jobs can't starve API
```

Real Examples:

- **Netflix**: Separate thread pools per service
- **Uber**: Isolate critical paths

Interview Usage:
"I'd use bulkhead pattern with separate thread pools for critical vs non-critical operations to prevent resource starvation"

---

44. Service Discovery

Definition: Automatically detect services in distributed system

Problem:

```
Microservices architecture:
How does Service A find Service B?
IPs change, instances scale up/down
```

Solution:

```
[Service Registry] (Consul/etcd)
    ↓
Stores: service-name → [IP1, IP2, IP3]
    ↓
Services register on startup
Services query registry to find others
```

Technologies:

- **Consul**: HashiCorp's service discovery
- **etcd**: Used by Kubernetes
- **Zookeeper**: Apache's coordinator
- **Eureka**: Netflix's service registry

Interview Usage:
"For microservices, I'd use Consul for service discovery so services can find each other dynamically"

---

45. Connection Pooling

Definition: Reuse database connections instead of creating new ones

Problem:

```
Creating new connection:
- TCP handshake
- Authentication
- Takes 10-100ms

1000 requests/second × 50ms = 50 seconds of wasted time!
```

Solution:

```
Connection Pool:
- Pre-create 20-50 connections
- Reuse connections
- Return to pool after use

1000 requests reuse same 20 connections
```

Benefits:

- Faster requests (no connection overhead)
- Limit concurrent connections to database
- Handle connection failures gracefully

Interview Usage:
"I'd use connection pool with 50 connections to avoid overhead of creating new database connections on every request"

---

46. Monitoring & Observability

Monitoring: Collect metrics about system health

Observability: Understand system state from outputs

Three Pillars:

1. Metrics (numbers):

```
- Request rate (QPS)
- Error rate (%)
- Latency (p50, p95, p99)
- CPU, memory, disk usage

Tools: Prometheus, Datadog, CloudWatch
```

2. Logs (events):

```
- Application logs
- Error logs
- Access logs
- Audit logs

Tools: ELK Stack (Elasticsearch, Logstash, Kibana)
```

3. Traces (request flow):

```
[Request] → [API Gateway] → [User Service] → [Database]
     5ms         10ms              20ms          30ms

Total: 65ms latency
Bottleneck: Database (30ms)

Tools: Jaeger, Zipkin, AWS X-Ray
```

Interview Usage:
"I'd use Prometheus for metrics, ELK for logs, and Jaeger for distributed tracing to identify bottlenecks"

---

47. Chaos Engineering

Definition: Deliberately inject failures to test system resilience

Netflix's Chaos Monkey:

```
Randomly terminates production instances
Forces engineers to design for failure
Reveals weaknesses before customers do
```

Types of Chaos:

- **Chaos Monkey**: Kill random instances
- **Chaos Gorilla**: Take down entire availability zone
- **Latency Monkey**: Inject artificial delays
- **Chaos Kong**: Take down entire region

Interview Usage:
"Netflix's Chaos Engineering approach ensures system handles failures gracefully by regularly testing fault tolerance in production"

---

48. Blue-Green Deployment

Definition: Zero-downtime deployment strategy

How It Works:

```
Blue (Current):  v1.0 serving 100% traffic
Green (New):     v2.0 deployed, serving 0% traffic

Test Green → Switch traffic → Green serves 100%
If problems → Switch back to Blue instantly
```

Benefits:

- Zero downtime
- Instant rollback
- Test in production before full deployment

Interview Usage:
"I'd use blue-green deployment so we can deploy new version without downtime and instantly rollback if issues arise"

---

49. Canary Deployment

Definition: Gradual rollout to subset of users

How It Works:

```
Step 1: Deploy to 5% of servers
Step 2: Monitor metrics
Step 3: If good → 25% → 50% → 100%
Step 4: If bad → Rollback 5%
```

Benefits:

- Catch bugs early
- Limited blast radius
- Real user testing

Interview Usage:
"I'd use canary deployment, rolling out to 5% of users first to catch issues before affecting everyone"

---

50. Data Lake

Definition: Centralized repository for raw data (structured + unstructured)

vs. Data Warehouse:

```
Data Lake:
- Raw, unprocessed data
- Schema-on-read
- Cheap storage (S3)
- For: Data science, exploration

Data Warehouse:
- Processed, structured data
- Schema-on-write
- Expensive (Redshift)
- For: Business intelligence
```

Example Flow:

```
[Application] → [Data Lake (S3)]
                      ↓
                [Process/Transform]
                      ↓
             [Data Warehouse (Redshift)]
                      ↓
             [BI Dashboard (Tableau)]
```

Interview Usage:
"I'd store raw events in S3 data lake, then process into Redshift warehouse for analytics"

Quick Reference

Most Important Terms for Interviews

Must Know (Use in every interview):

1. ✅ Horizontal vs Vertical Scaling
2. ✅ Load Balancer (Layer 4 vs Layer 7)
3. ✅ Caching (Cache-aside pattern)
4. ✅ Database Replication (Master-Slave)
5. ✅ Sharding
6. ✅ CAP Theorem (CP vs AP)
7. ✅ CDN
8. ✅ Message Queue (Kafka)
9. ✅ Microservices
10. ✅ Rate Limiting

Should Know (Mention when relevant):
11. ✅ Consistent Hashing

12. ✅ CQRS
13. ✅ Event Sourcing
14. ✅ Circuit Breaker
15. ✅ WebSocket
16. ✅ Idempotency
17. ✅ Fan-out (Push vs Pull)
18. ✅ Bloom Filter
19. ✅ ACID vs BASE
20. ✅ Denormalization

Nice to Know (For senior roles):
21. Back Pressure
22. Bulkhead Pattern
23. Chaos Engineering
24. Geohashing
25. Service Discovery

Interview Cheat Sheet

When to Mention Each Term

"Design Twitter":

- Sharding (by tweet_id)
- Fan-out (hybrid push-pull)
- Redis Cache (timelines)
- Cassandra (write-heavy tweets)
- Message Queue (async processing)
- CDN (media delivery)

"Design Instagram":

- S3 + CDN (photo storage)
- Cassandra (photo metadata)
- Redis Cache (feed)
- Fan-out (posts to followers)
- Geohashing (location tags)

"Design Uber":

- Geohashing (driver matching)
- WebSocket (real-time location)
- ACID (payment transactions)
- Redis Geo (active drivers)
- Sharding (geographic)

"Design URL Shortener":

- Consistent Hashing (distributed counter)
- Base62 Encoding (short codes)
- Redis Cache (hot URLs)
- Rate Limiting (prevent abuse)
- Bloom Filter (check existence)

"Design WhatsApp":

- WebSocket (real-time messaging)
- End-to-end encryption
- Cassandra (message storage)
- Push Notifications
- CAP Theorem (AP for availability)

Terminology Combinations

Powerful Phrases for Interviews

"To achieve **high availability** with **fault tolerance**, I'd use **master-slave replication** across **3 availability zones** with **automatic failover**"

"For **scalability**, I'd start with **vertical scaling**, then move to **horizontal scaling** with **load balancer** and **database sharding** when traffic exceeds **10K QPS**"

"To handle **read-heavy workload**, I'd implement **cache-aside** pattern with **Redis**, achieving **80% cache hit ratio** and reducing **database load** by **5x**"

"For **write-heavy** tweets (**500M per day**), I'd use **Cassandra** with **consistent hashing** for **linear scalability** and **geographic distribution**"

"To prevent **cascading failures**, I'd implement **circuit breaker** pattern and **bulkhead isolation** with **graceful degradation** for non-critical features"

Common Interview Questions & Terms to Use

Q: "How would you scale this system?"

Answer Template:

```
"I'd scale through:

1. Horizontal Scaling: Add more application servers behind load balancer
2. Database Replication: Master-slave with 5 read replicas
3. Caching: Redis with 80-20 rule, achieving 80% cache hit ratio
4. CDN: For static assets and media files
5. Sharding: When single DB can't handle writes (>10K QPS)
6. Message Queue: For async processing (Kafka)
```

```
7. Microservices: Split into independently scalable services
"
```

Q: "How would you ensure high availability?"

Answer Template:

```
"I'd ensure HA through:

1. Multi-Region Deployment: Deploy across 3 regions
2. Redundancy: 3 replicas of all data
3. Load Balancing: Active-active with health checks
4. Failover: Automatic with < 30 second RTO
5. Circuit Breaker: Prevent cascading failures
6. Graceful Degradation: Core features work even if some components fail
7. Monitoring: Prometheus + PagerDuty for alerts
"
```

Q: "How would you improve performance?"

Answer Template:

```
"I'd improve performance by:

1. Caching: Redis for hot data (< 1ms latency)
2. CDN: For media files (10x faster)
3. Indexing: Database indexes for common queries
4. Denormalization: Avoid expensive JOINs
5. Connection Pooling: Reuse database connections
6. Async Processing: Message queue for non-blocking operations
7. Compression: Gzip API responses (70% smaller)
"
```

Final Checklist

Before finishing any system design answer, mention:

- ✅ **Scalability**: How system scales (horizontal, sharding, caching)
- ✅ **Availability**: How to achieve 99.99% (replication, failover)
- ✅ **Performance**: Latency targets (< 100ms p99)
- ✅ **Consistency**: Strong vs eventual (CAP theorem choice)
- ✅ **Monitoring**: How to detect issues (metrics, logs, traces)
- ✅ **Security**: Rate limiting, authentication, encryption

Summary: The Power Words

Use these terms to sound knowledgeable:

Scalability: Horizontal scaling, sharding, load balancer, CDN
Performance: Caching, indexing, denormalization, CDN
Reliability: Replication, failover, circuit breaker, graceful degradation
Consistency: ACID, BASE, CAP theorem, eventual consistency
Architecture: Microservices, CQRS, event sourcing, API gateway
Data: Sharding, consistent hashing, replication, federation

Document Version: 1.0
Last Updated: January 8, 2025
For: System Design Interview Preparation
Status: Complete & Interview-Ready ✅