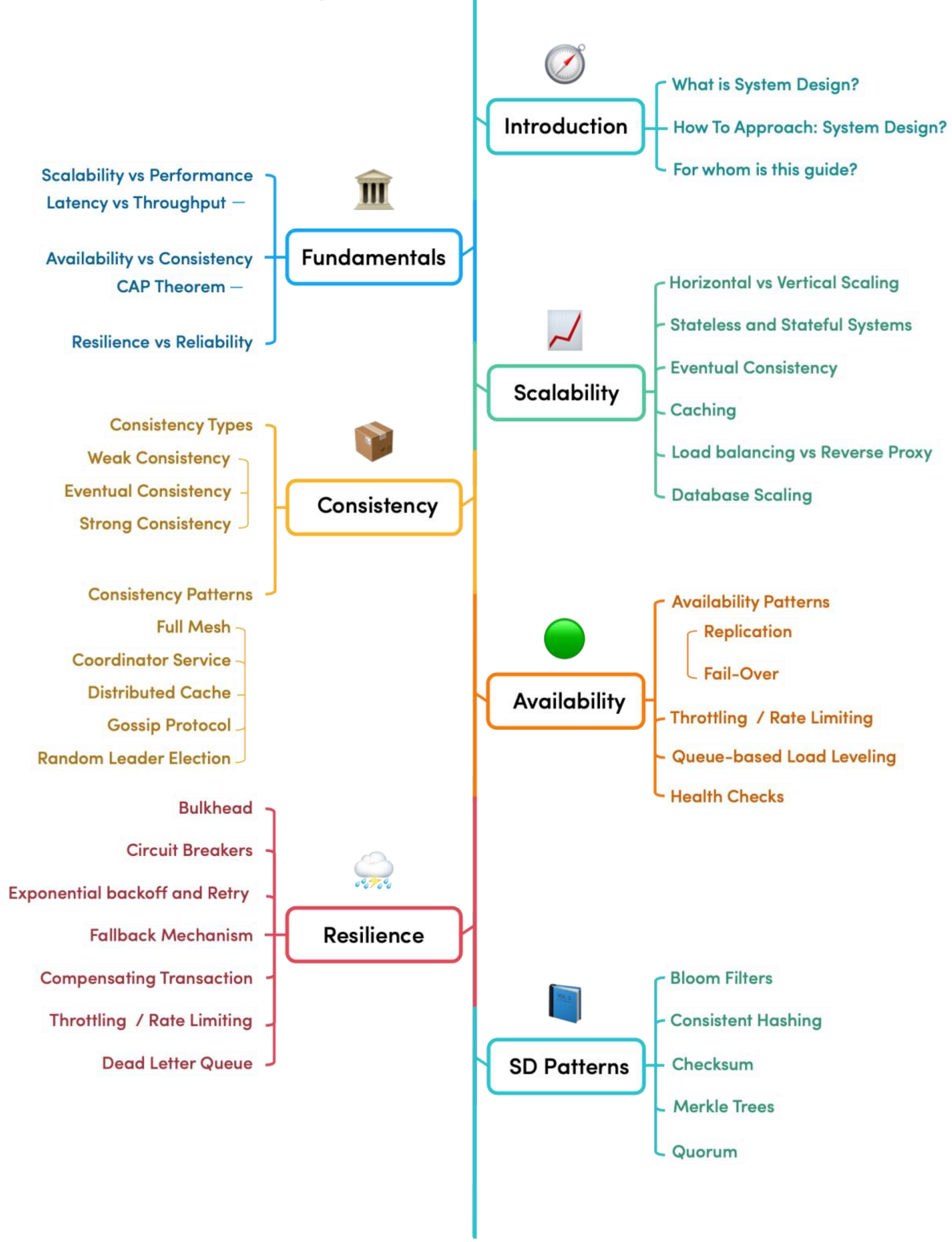
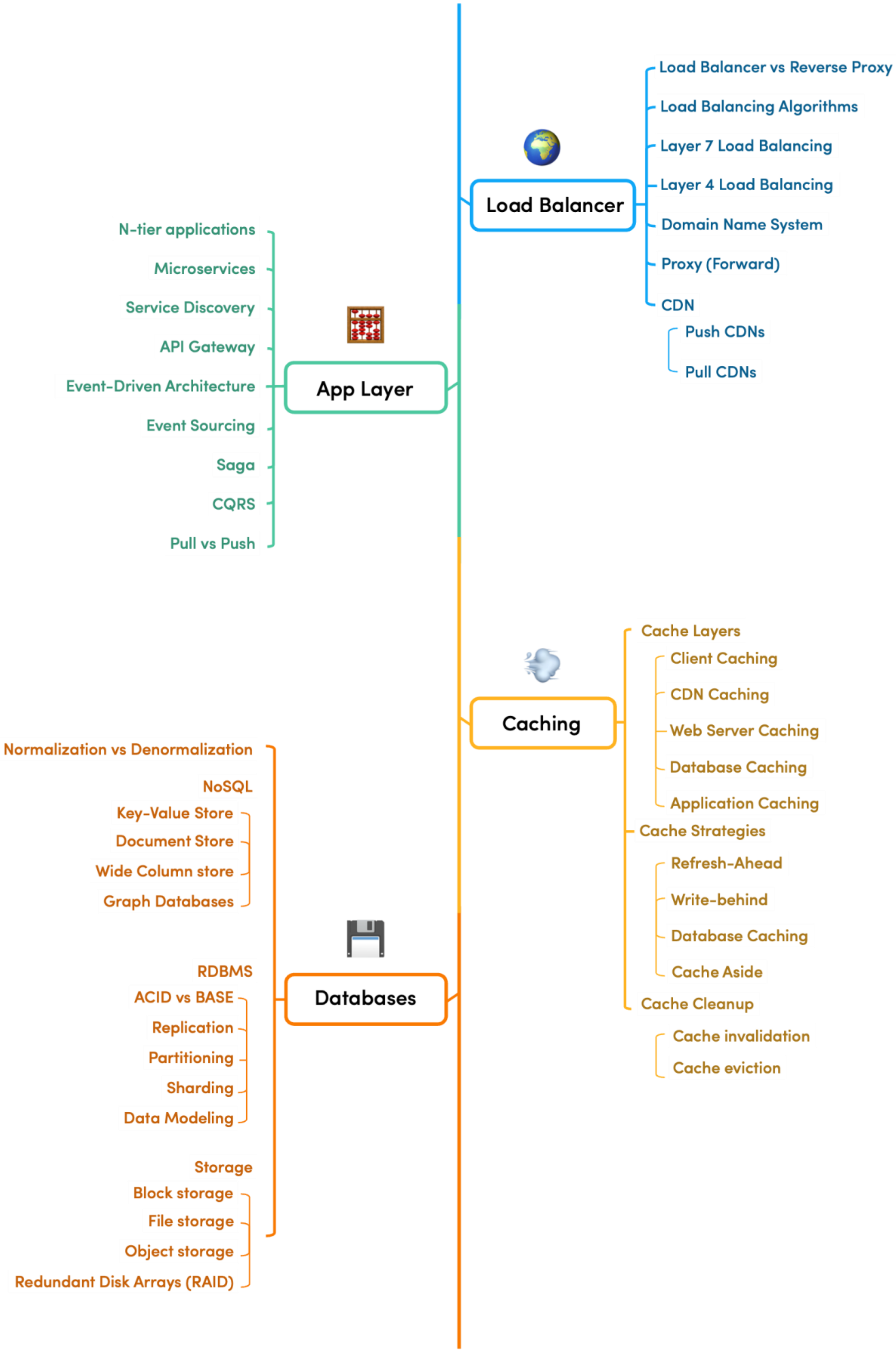
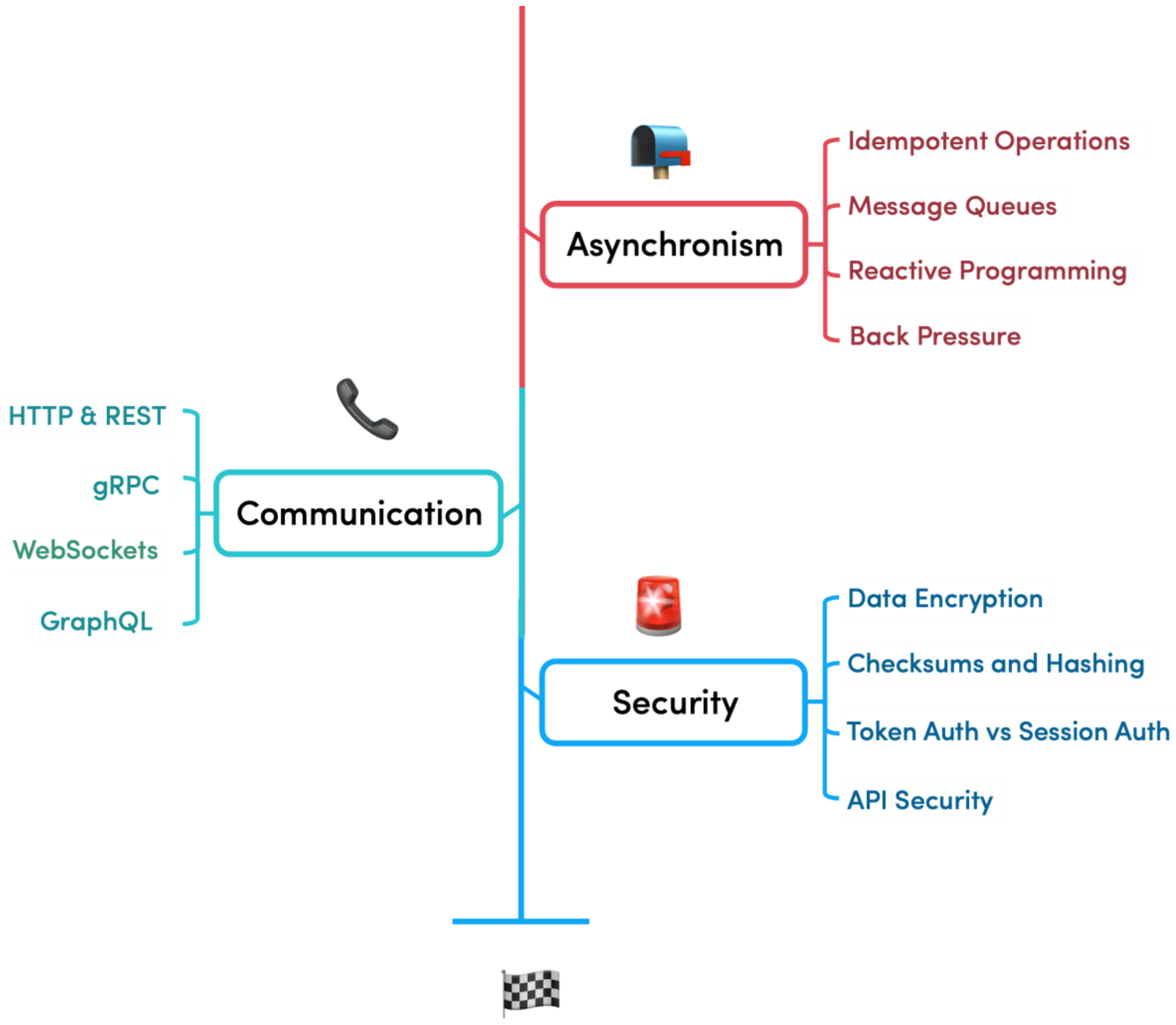
# System Design Road Map



- Introduction
  - What is System Design?
  - How To Approach: System Design?
  - For whom is this guide?
- Fundamentals
  - Scalability vs Performance
  - Latency vs Throughput
  - Availability vs Consistency
  - CAP Theorem
  - Resilience vs Reliability
- Scalability
  - Horizontal vs Vertical Scaling
  - Stateless and Stateful Systems
  - Eventual Consistency
  - Caching
  - Load balancing vs Reverse Proxy
  - Database Scaling
- Consistency
  - Consistency Types
    - Weak Consistency
    - Eventual Consistency
    - Strong Consistency
  - Consistency Patterns
    - Full Mesh
    - Coordinator Service
    - Distributed Cache
    - Gossip Protocol
    - Random Leader Election
- Availability
  - Availability Patterns
    - Replication
    - Fail-Over
  - Throttling / Rate Limiting
  - Queue-based Load Leveling
  - Health Checks
- Resilience
  - Bulkhead
  - Circuit Breakers
  - Exponential backoff and Retry
  - Fallback Mechanism
  - Compensating Transaction
  - Throttling / Rate Limiting
  - Dead Letter Queue
- SD Patterns
  - Bloom Filters
  - Consistent Hashing
  - Checksum
  - Merkle Trees
  - Quorum



- Load Balancer
  - Load Balancer vs Reverse Proxy
  - Load Balancing Algorithms
  - Layer 7 Load Balancing
  - Layer 4 Load Balancing
  - Domain Name System
  - Proxy (Forward)
  - CDN
    - Push CDNs
    - Pull CDNs
- App Layer
  - N-tier applications
  - Microservices
  - Service Discovery
  - API Gateway
  - Event-Driven Architecture
  - Event Sourcing
  - Saga
  - CQRS
  - Pull vs Push
- Caching
  - Cache Layers
    - Client Caching
    - CDN Caching
    - Web Server Caching
    - Database Caching
    - Application Caching
  - Cache Strategies
    - Refresh-Ahead
    - Write-behind
    - Database Caching
    - Cache Aside
  - Cache Cleanup
    - Cache invalidation
    - Cache eviction
- Databases
  - Normalization vs Denormalization
  - NoSQL
    - Key-Value Store
    - Document Store
    - Wide Column store
    - Graph Databases
  - RDBMS
    - ACID vs BASE
    - Replication
    - Partitioning
    - Sharding
    - Data Modeling
  - Storage
    - Block storage
    - File storage
    - Object storage
    - Redundant Disk Arrays (RAID)



📫 Asynchronism

- Idempotent Operations
- Message Queues
- Reactive Programming
- Back Pressure

📞 Communication

- HTTP & REST
- gRPC
- WebSockets
- GraphQL

🚨 Security

- Data Encryption
- Checksums and Hashing
- Token Auth vs Session Auth
- API Security

🏁

# INTRODUCTION

## What is System Design?

System design is basically the process of figuring out all the parts of a system and how they work together to meet certain requirements. It's like solving a big puzzle, where you break down a problem into smaller parts and design each part to work seamlessly with the others.

When it comes to software engineering, system design is a super important phase in creating a software system. This is where you figure out the overall design and architecture of the system, as well as its components.

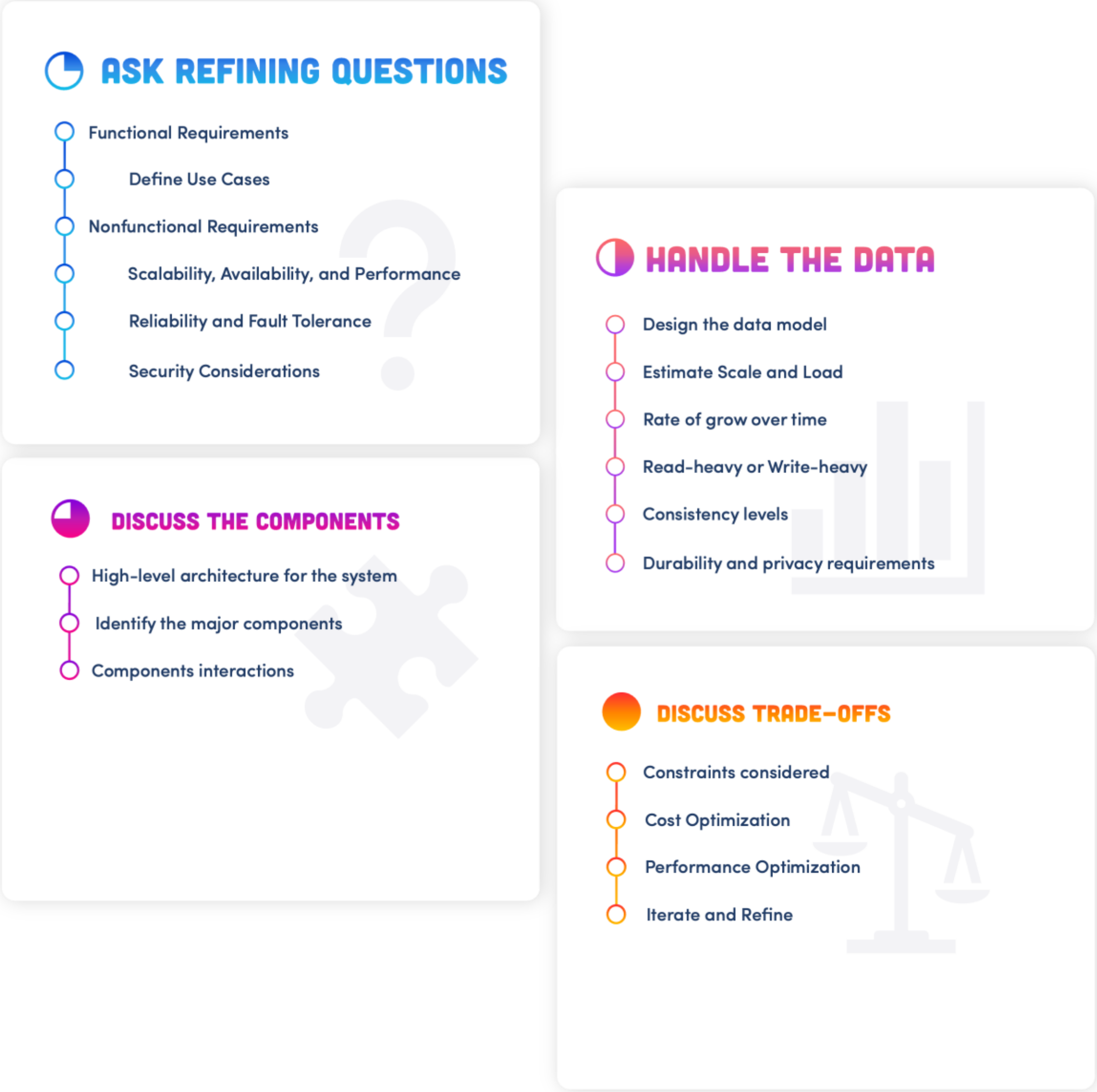
Oh, and heads up: if you're interviewing for a software engineering job, you'll probably have to do a system design interview round. That means you'll have to design a whole system from scratch and talk about your choices along the way. It's all about finding the right balance and weighing the pros and cons of each option.

System design is an iterative process, meaning you'll be testing and refining your design multiple times until it meets all the requirements. You'll also take a look at any current systems in place and see what's working and what's not.

```
+--------------------+
| Gather Requirements |
+--------------------+
       |
       V
+----------------+
| Handle the Data |
+----------------+
       |
       V
+-----------------+
| Define Components |
+-------------------+
       |
       V
+---------------------+
| Identify Trade-offs |
+---------------------+
```

## How To Approach: System Design?

When you're diving into a system design puzzle, here's a chill list of steps to follow:



ASK REFINING QUESTIONS

- Functional Requirements
  - Define Use Cases
- Nonfunctional Requirements
  - Scalability, Availability, and Performance
  - Reliability and Fault Tolerance
  - Security Considerations

HANDLE THE DATA

- Design the data model
- Estimate Scale and Load
- Rate of grow over time
- Read-heavy or Write-heavy
- Consistency levels
- Durability and privacy requirements

DISCUSS THE COMPONENTS

- High-level architecture for the system
- Identify the major components
- Components interactions

DISCUSS TRADE-OFFS

- Constraints considered
- Cost Optimization
- Performance Optimization
- Iterate and Refine

Understand the Problem:

First, wrap your head around what you're building. No rocket science here, just get what it's all about.

Gather Requirements:

Chat with folks involved and collect all the "wants" and "needs." Think of it as making a wish list for your project.

Scalability:

Ensure your system can grow without breaking a sweat. Imagine it like hosting a party and making sure you have enough snacks for everyone!

Define Key Components:

Break down your system into smaller parts. It's like assembling a jigsaw puzzle - each piece has a role.

Data Flow Diagrams:

Visualize how data will move within your system. Imagine it as a flowchart for your digital river.

Database Design:

Organize your data smartly. Think of it as arranging your books on shelves so you can find them later.

System Architecture:

Plan how everything will fit together. It's like sketching a rough blueprint for your dream house.

Security:

Guard your system like a fort. Keep out unwanted visitors and protect your precious data.

Testing and Optimization:

Tinker, test, and tune. Make sure everything runs like a well-oiled machine.

Review and Iterate:

Check your work and keep improving. It's like refining a recipe to make it perfect.

Remember, system design is like building your very own digital playground - it's an art and a science mixed with a dash of creativity and lots of problem-solving fun! 🎨🛠🚀

## For whom is this guide?

This guide is for anyone who wants to learn about designing scalable systems. Whether you're a software engineer, backend developer, or a product/project managers, this guide has got you covered!

It's also perfect for those prepping for system design interviews, as it gives you a comprehensive understanding of all the key concepts and considerations involved in the design process. System design is essential for back-end developers aspiring to become principal engineers or solution architects.

We mentioned a wide range of topics, with explanations and external links for those who want to dive deeper.

# FUNDAMENTALS

## Performance vs Scalability

### Performance:

-> -> -> -> ->

---- ---- ---- ----

**Definition**: Performance refers to how well a system or component accomplishes its tasks within a given set of constraints, such as time, resources, or user expectations.

**Focus**: Performance focuses on optimizing the speed and responsiveness of individual tasks, operations, or requests within the system.

**Measurement**: Performance is often measured by metrics like response time, latency, throughput, and resource utilization.

**Goal**: The goal of improving performance is to ensure that the system's operations are completed quickly and efficiently, providing a smooth and responsive user experience.

### Scalability:

**Definition**: Scalability refers to a system's ability to handle increased load or demand by adapting its capacity, resources, or components.

**Focus**: Scalability focuses on a system's ability to accommodate growth without sacrificing performance or availability.

**Measurement**: Scalability is often measured by observing how well the system maintains performance as load or the number of users increases.

**Goal**: The goal of achieving scalability is to ensure that the system can handle increased workloads or traffic without significant degradation in performance, and it can do so by adding resources (scaling out) or increasing the capacity of existing resources (scaling up).

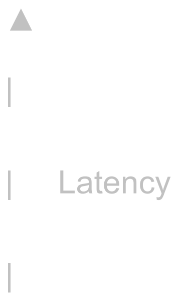
In summary, while both performance and scalability aim to improve the efficiency and effectiveness of a system, they address different aspects.

Here's another perspective on performance versus scalability:

- If your system is slow for a single user, you have a performance issue.
- If your system is fast for a single user but slows down when under heavy load, you have a scalability issue.

## Latency vs Throughput

### Latency:



▼

Latency, often referred to as response time, is the amount of time it takes for a single request to travel from the sender to the receiver and back, including any processing time along the way. It is essentially the delay experienced by a single operation or request. Low latency is generally desirable, as it means that requests are being handled quickly and users are getting timely responses.

- **Example**: Consider a web server receiving requests from clients. The latency for a particular request is the time it takes for the server to process the request and send a response back to the client.

### Throughput:

```
--> --> --> -->
```

Throughput, on the other hand, refers to the number of operations or requests that a system can handle in a given amount of time. It is a measure of the system's processing capacity. High throughput indicates that the system can handle a large volume of operations effectively.

- **Example**: A database system might be able to handle 1000 queries per second. This means its throughput is 1000 queries per second.

## Availability vs Consistency

### Availability:

```
╔══════════════════╗
║                  ║
║      System      ║
║   Operational    ║
║                  ║
╚══════════════════╝
```

Availability refers to the ability of a system to remain operational and accessible to users, even in the presence of failures. An available system ensures that users can still interact with it and receive responses, even if some parts of the system are experiencing issues. High availability systems are designed to minimize downtime and provide a reliable user experience.

- **Example**: A highly available web application continues to serve user requests even if one of its servers goes down. The system uses redundancy and failover mechanisms to ensure that users experience minimal disruption.

## Consistency:

```
╔══════════════════════════════╗
║    /\     /\    ║
║   /  \   /  \   ║
║  /    \ /    \  ║
╚══════════════════════════════╝
```

Consistency refers to the requirement that all nodes in a distributed system see the same data at any given point in time, regardless of where the data is accessed. In other words, updates made to the data in one part of the system should eventually be reflected in all other parts of the system. Maintaining consistency can be challenging in distributed systems, especially in the face of network partitions and failures.

- **Example**: A distributed database system needs to ensure that when a data update is confirmed, all subsequent read requests for that data from any part of the system return the updated value.

It's important to note that there's a trade-off between availability and consistency, often referred to as the CAP theorem (Consistency, Availability, Partition Tolerance).

Systems that prioritize high availability may sacrifice consistency, while systems that prioritize consistency may sacrifice availability.

## CAP Theorem

According to CAP theorem, in a distributed system, you can only support two of the following guarantees:

```
    C
   / \
  /   \
 /     \
A-------P
```

### Consistency:

This means everyone sees the same data at the same time. Like in a game, if you shoot a monster, everyone playing should see the monster getting hit right away.

### Availability:

This means your system keeps working and doesn't crash, even if something goes wrong. Like in the game, even if one part of the game slows down, you can still play in other parts.

### Partition Tolerance:

This means your system can handle things like the internet having problems and parts of your system not being able to talk to each other for a bit. It's like your game still working even if some players can't talk to each other for a moment.

Here are the three combinations:

### CA:

You choose Consistency and Availability. This means everyone sees the same data, and the system keeps working well. But if there's a problem with the network, your system might slow down or stop working.

### CP:

You choose Consistency and Partition Tolerance. This means everyone sees the same data, even if some parts of the system can't talk to each other for a bit. But during that time, your system might slow down.

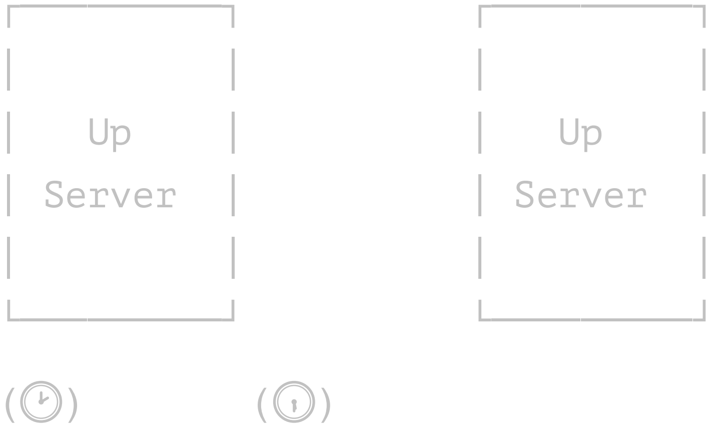
### AP:

You choose Availability and Partition Tolerance. This means your system keeps working even if the network has issues. But because different parts of the system might not have the exact same data right away, things might seem a little inconsistent for a short time.

So, the CAP theorem helps you decide what's most important for your system when things get tough: having everyone see the same data, keeping the system working, or handling problems in the network. You can't have all three perfect at once, so you need to choose what fits best for what you're building.

## Reliability vs Resilience

### Reliability:



Reliability refers to a system's ability to consistently perform its intended function without failure over a specified period of time. A reliable system minimizes downtime and ensures consistent performance expectations. In essence, reliability is about a system's ability to avoid failures and maintain functionality.

- **Example**: A reliable web server consistently responds to user requests without crashing or experiencing errors. It maintains a high level of availability and consistency.

### Resilience:

```
┌──────────┐        ┌──────────┐
 |          |         |          |
 |  Failed  |         |    Up    |
 |  Server  |   ->    |  Server  |
 |          |         |          |
 └──────────┘         └──────────┘

 (🕑)        (🔒)
```

Resilience refers to the ability of a system to recover from failures or adverse conditions and continue functioning, albeit possibly at a reduced capacity or with some temporary disruptions. A resilient system is designed to absorb shocks, adapt to changes, and bounce back from failures with minimal impact.

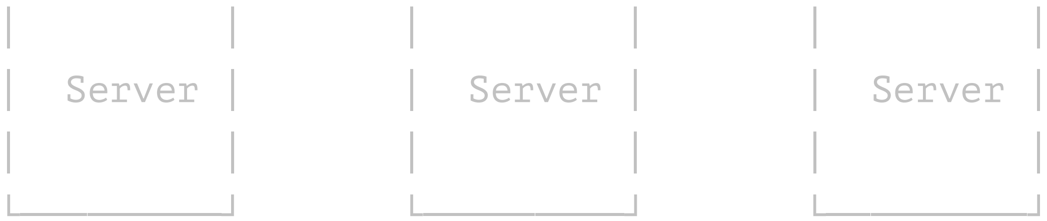
- **Example**: A resilient cloud-based application can automatically shift its workload to different servers if one server fails, ensuring that users experience minimal interruption even in the face of failures.

# SCALABILITY

## Horizontal Scaling vs Vertical Scaling

### Horizontal Scaling (Scaling Out):

```
 ┌──────────┐      ┌──────────┐      ┌──────────┐
```

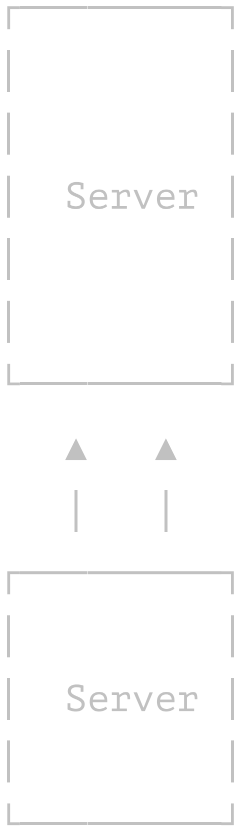


Horizontal scaling involves adding more machines or nodes to a system to handle increased load. This approach focuses on distributing the workload across multiple machines, which can help increase the overall capacity of the system. Horizontal scaling is often used in distributed systems and cloud environments.

In the context of system design:

- **Example:** If a website is experiencing high traffic, you might add more web servers to the system. Each server handles a portion of the incoming requests, allowing the system to collectively handle a higher number of users.

## Vertical Scaling (Scaling Up):



Vertical scaling involves upgrading the resources of an existing machine to handle increased load. This approach focuses on improving the capacity of a single machine by adding more processing power, memory, storage, etc.

In the context of system design:

- **Example**: If a database server is struggling with heavy data processing, you might upgrade its CPU and memory to enhance its processing capabilities.

## What to chose?

The choice between these two approaches depends on factors like the nature of the application, available resources, budget, and scalability goals. Horizontal scaling is often favored in modern cloud-based and distributed systems, as it provides greater flexibility and fault tolerance by distributing the workload across multiple machines.

# Stateless Systems vs Stateful Systems

## Stateless Systems:

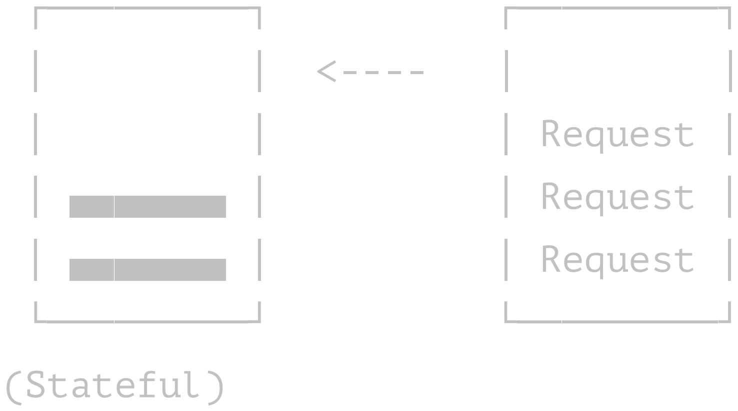
```
  ┌─────────┐           ┌─────────┐
  │         │   <----   │ Request │
  │         │           │ Request │
  │         │           │ Request │
  └─────────┘           └─────────┘

(No State)
```

A stateless system is designed in a way that each request from a client contains all the information necessary for the server to fulfill that request. The server doesn't rely on any previous interactions or stored data to process the current request. This means that each request is independent, and the server doesn't maintain any memory of past requests.

In the context of system design:

- **Example:** A web server that serves static content like images or CSS files is typically stateless. It simply responds to incoming requests without requiring any information about previous requests.

## Stateful Systems



A stateful system, on the other hand, maintains and relies on state information about past interactions. It keeps track of user sessions, historical data, or other contextual information that's relevant to fulfilling requests. Stateful systems often store this information in databases, caches, or other persistent storage.

In the context of system design:

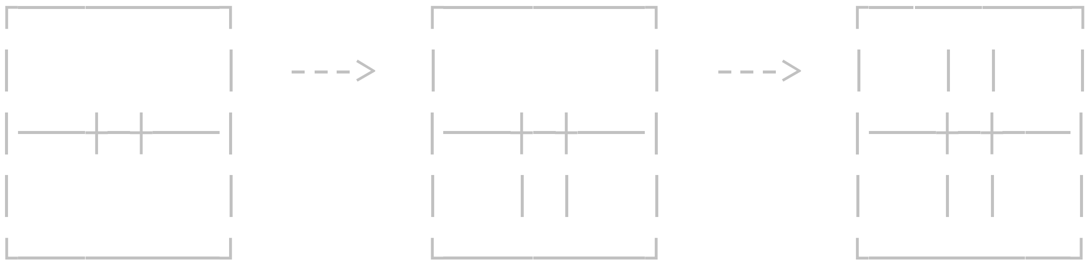
- **Example:** An e-commerce application that tracks users' shopping carts would be stateful. The server needs to remember the contents of a user's cart even as they navigate between different pages.

## What to chose?

Choosing between a stateless and a stateful approach depends on the specific requirements of the application.

Stateless systems are often preferred in modern microservices architectures for their scalability and simplicity, while stateful systems are chosen when applications need to maintain context or state between interactions, even though they can introduce more complexity in terms of data management and synchronization.

# Eventual consistency



Eventual consistency is a principle in distributed systems that describes a level of data consistency where, given enough time and under certain conditions, all replicas of a piece of data in a distributed system will converge to the same value. In other words, while immediate consistency might not be guaranteed, the system ensures that, eventually, all copies of data will become consistent.

This concept is particularly relevant when dealing with large-scale distributed systems where maintaining strong consistency across all nodes in real-time can be challenging due to factors like network delays, node failures, and high traffic. Eventual consistency offers a trade-off between immediate consistency and system performance and availability.

Eventual consistency can help with **scalability** by allowing the system to:

## Handle High Traffic and Load:

In systems with a large number of users or high levels of traffic, maintaining immediate consistency across all nodes can lead to performance bottlenecks. Eventual consistency relaxes this requirement, allowing nodes to continue processing requests without waiting for synchronization in every case.

## Reduce Latency:

Achieving strong consistency in a distributed system often requires coordination and communication between nodes, which can introduce latency. Eventual consistency reduces the need for constant synchronization, helping to keep response times lower.

## Tolerate Network Delays and Failures:

In distributed systems, network partitions and communication failures can occur. Eventual consistency allows nodes to continue functioning even when communication between them is temporarily disrupted.

### Improve Availability:

By relaxing the requirement for immediate consistency, systems can remain operational even when a subset of nodes experiences issues. This improves overall system availability and fault tolerance.

It's important to note that eventual consistency might not be suitable for all types of applications. Systems that require strong consistency guarantees for critical operations (like financial transactions) might not benefit from the eventual consistency model. However, many applications, such as social media feeds, recommendation engines, and analytics platforms, can leverage eventual consistency to achieve high levels of scalability without compromising the user experience.

## Caching

```
┌────────────┐        ┌────────────┐        ┌────────────┐
  |            |        |   Cache    |        |            |
  |   Client   |  --->  |            |  --->  |   Server   |
  |            |        |            |        |            |
  └────────────┘        └────────────┘        └────────────┘
```

Caching is a technique used in system design to improve the performance and scalability of applications by storing frequently accessed data in a temporary storage area called a cache. The primary purpose of caching is to reduce the need to retrieve data from slower and more resource-intensive sources, such as databases or remote services, by serving it directly from the cache.

Caching helps with **scalability** in several ways:

### Improved Response Time:

By serving data from a cache, the system can respond to requests much faster than if it had to fetch the data from a slower data source. This reduces the perceived latency and improves the user experience.

### Reduced Load on Backend Systems:

Caching offloads the workload from the primary data source (like a database or an API) by serving frequently accessed data from the cache. This reduces the load on the backend systems, allowing them to handle more requests efficiently.

### Enhanced Scalability:

Caching can be especially effective in distributed systems and microservices architectures. By reducing the load on shared resources, caching allows each individual service to scale more easily without putting too much strain on the backend systems.

### Minimized Network Traffic:

Caching can significantly reduce the amount of data transferred over the network. Since cached data is often located closer to the application, there's less need to fetch data over longer distances.

### Better Resource Utilization:

Caching helps optimize resource utilization by reducing the need to perform repetitive and resource-intensive operations, such as complex calculations or database queries, multiple times.

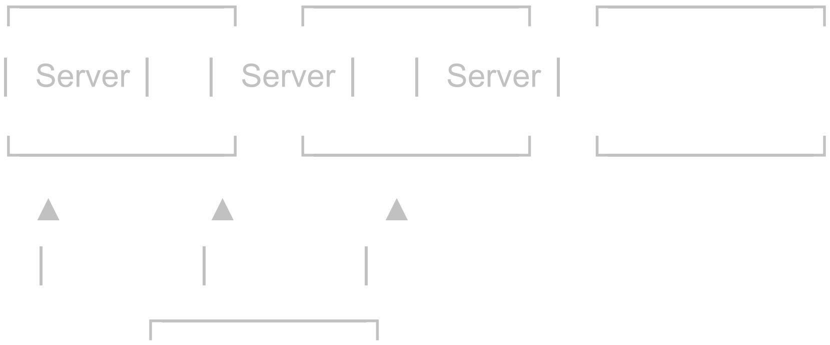
### High Availability During Failures:

Caches can store copies of data that are less likely to change frequently. During outages or failures of backend systems, cached data can help maintain availability and prevent service disruptions.

It's important to note that caching requires careful management to ensure that cached data remains accurate and up to date. Strategies like cache expiration, cache eviction, and cache invalidation are used to maintain the integrity of cached data.

# Load Balancing vs Reverse Proxy

## Load Balancing



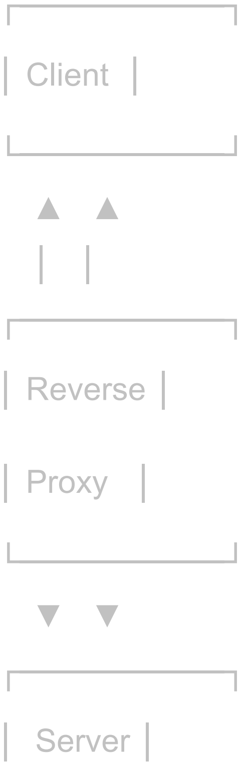
Load balancing is the practice of distributing incoming network traffic across multiple servers or resources to ensure that no single server becomes overwhelmed and that the workload is efficiently shared. The primary goal of load balancing is to improve the performance, availability, and scalability of a system by evenly distributing traffic and preventing any individual server from becoming a bottleneck.

Load balancers can operate at different layers of the network stack, including the transport (Layer 4) and application (Layer 7) layers. Load balancers monitor the health and capacity of the backend servers and direct incoming requests to the least busy server or the one with the best performance metrics.

In the context of system design:

- Example: A website receiving a large number of user requests might use a load balancer to distribute those requests among multiple web servers. This ensures that each server handles a manageable amount of traffic, improving response times and preventing server overloads.

## Reverse Proxy



A reverse proxy is a server that sits between client devices and backend servers. It handles incoming requests from clients, forwards those requests to appropriate backend servers, and then returns the response to the clients. Reverse proxies often provide additional features like caching, security, SSL termination, and content compression.

Reverse proxies are commonly used to improve security, performance, and manageability. They can offload tasks from backend servers, centralize access control, and optimize network traffic by serving cached content directly to clients.

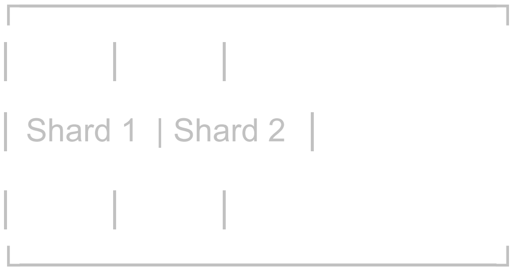
In the context of system design:

- **Example:** An organization might deploy a reverse proxy in front of their web application servers. The reverse proxy can handle SSL encryption and decryption, distribute requests to backend servers using load balancing, and cache frequently accessed content to reduce the load on application servers.

# Database Scaling

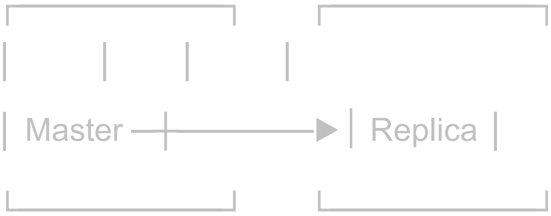
There are several main techniques to scale a database, each with its own benefits, challenges, and considerations.

## Database Sharding:



Sharding involves partitioning the data into smaller subsets, called shards, and distributing these shards across multiple servers. Each server holds a portion of the data, reducing the load on individual servers and improving performance. Sharding requires careful planning to ensure that data is evenly distributed, and query routing mechanisms are in place to direct queries to the appropriate shard.

## Replication with Load Balancing:



Replication involves creating copies (replicas) of the database on multiple servers. Each replica can handle read requests, distributing the read workload and improving performance. Load balancers distribute incoming requests across the replicas, ensuring even distribution of traffic.

## Data Partitioning:

Data partitioning involves breaking a single table into smaller partitions based on a certain criterion (e.g., range, hash, list). Each partition resides on a separate server, allowing for efficient data storage and retrieval. This technique can improve query performance and maintenance.

## Vertical Scaling (Scaling Up):

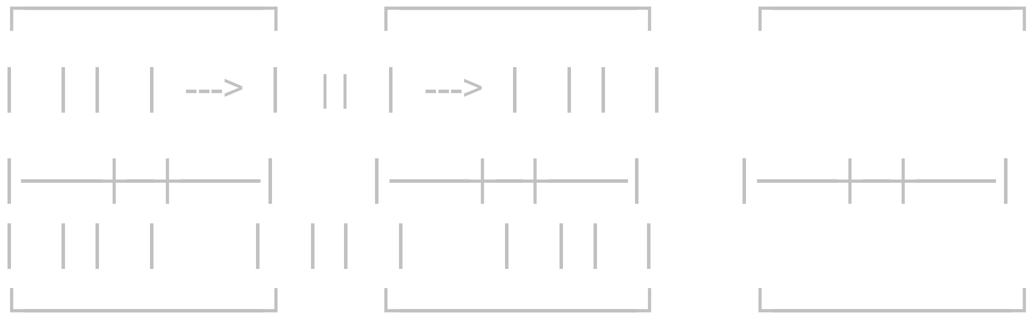
This involves upgrading the hardware resources of a single database server to handle increased workload. Common upgrades include adding more CPU cores, increasing memory, and expanding storage capacity. Vertical scaling is relatively straightforward but has limitations based on the maximum resources a single server can provide.

# CONSISTENCY

## Consistency Types

Consistency patterns refer to different levels of data consistency in distributed systems. Each pattern provides a trade-off between data correctness, availability, and performance. Let's explore each consistency pattern:

### Strong Consistency:



Strong consistency ensures that all nodes in a distributed system see the same data at the same time. This means that once a write operation is confirmed, all subsequent read operations will return the updated data. Strong consistency provides a guarantee that there is no time lag or discrepancy in the data seen by different nodes.

When to use strong consistency:

- Use cases where data integrity is paramount, such as financial transactions or critical updates.
- Scenarios where maintaining data correctness is more important than high availability or low latency.
- Applications where users expect real-time consistency across all interactions.

## Weak Consistency:

```
┌──────────┐        ┌──────────┐        ┌──────────┐
│          │  --->  │          │  --->  │          │
│────┼─┼───│        │────┼─┼───│        │────┼─┼───│
│          │        │    │ │   │        │    │ │   │
└──────────┘        └──────────┘        └──────────┘
```

Weak consistency allows for a certain level of inconsistency in data views across different nodes. Nodes might see different versions of data for some time after an update, but eventually, all nodes will converge to a consistent state. Weak consistency provides higher availability and performance compared to strong consistency but allows for temporary inconsistencies.

When to use weak consistency:

- Applications where high availability and low latency are top priorities.
- Systems that can tolerate minor inconsistencies for a short duration, and where eventual consistency is acceptable.
- Use cases like social media feeds, where immediate consistency is not critical, and users are okay with seeing slightly different data for a brief period.

## Eventual Consistency:

```
┌──────────┐        ┌──────────┐        ┌──────────┐
│          │  --->  │          │  --->  │    │ │   │
│────┼─┼───│        │────┼─┼───│        │────┼─┼───│
│          │        │    │ │   │        │    │ │   │
└──────────┘        └──────────┘        └──────────┘
```

Eventual consistency is a relaxed form of consistency that guarantees that, given enough time and under specific conditions, all nodes will eventually converge to the same data state. Eventual consistency allows for temporary data divergence but ensures that the system eventually reaches a consistent state without the need for real-time synchronization.

When to use eventual consistency:

- Large-scale distributed systems where immediate consistency is challenging to achieve due to network delays or failures.

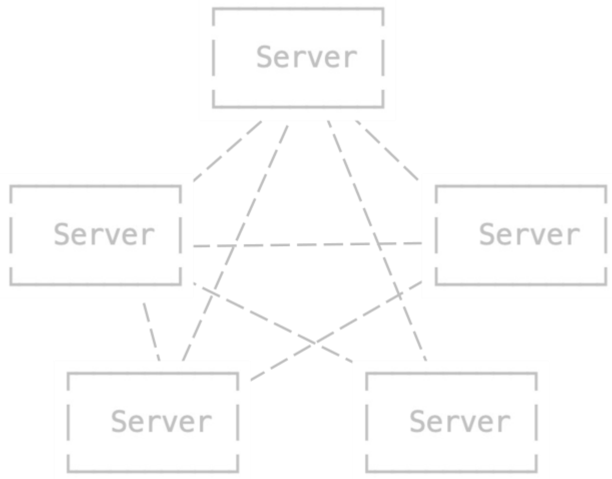
- Applications that prioritize high availability, fault tolerance, and scalability over immediate data correctness.
- Systems that can handle temporary data inconsistencies and reconcile differences over time.

## Consistency Patterns

The consistency patterns you mentioned are approaches or techniques used in distributed systems to manage data consistency, communication, and coordination among nodes.

Each pattern addresses specific challenges and scenarios. Let's explore each one:

### Full Mesh:

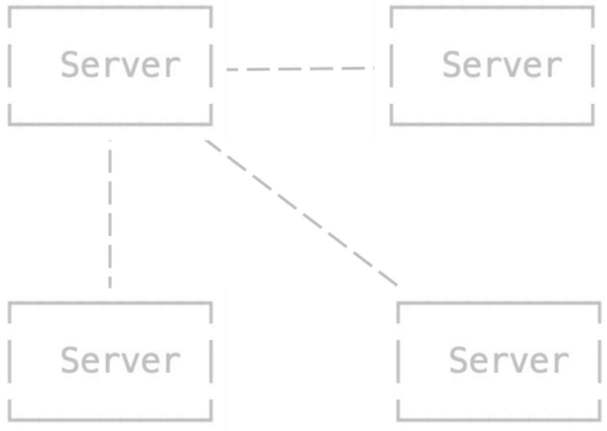


In a full mesh topology, every node directly communicates with every other node in the system. This approach facilitates direct communication and data sharing between nodes. While it provides strong connectivity, it can lead to increased network traffic and complexity as the number of nodes grows.

When to use the full mesh pattern:

- Smaller systems with a manageable number of nodes where direct communication is feasible.
- When strong coordination and data sharing are required between all nodes.

## Coordinator Service:

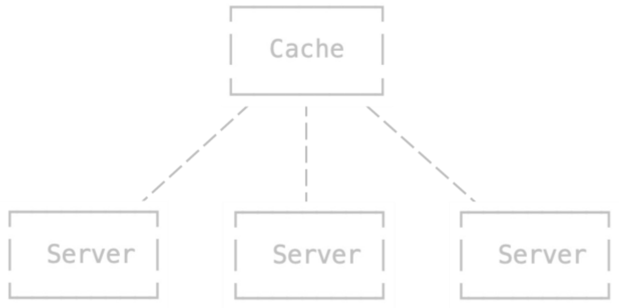


A coordinator service acts as a central point of control and coordination for distributed operations. Nodes communicate with the coordinator to perform tasks, acquire locks, or obtain information. The coordinator orchestrates actions and ensures consistency across the system.

When to use the coordinator service pattern:

- When you need a single point of coordination to manage distributed operations, such as distributed locking or leader election.
- For scenarios where a centralized decision-making process is acceptable.

## Distributed Cache:

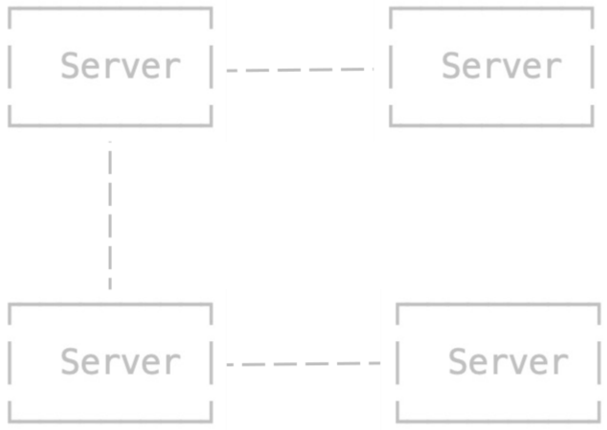


A distributed cache stores frequently accessed data in memory across multiple nodes. This pattern improves data retrieval performance by reducing the need to fetch data from slower storage systems like databases. Nodes can retrieve data directly from the cache, improving response times.

When to use the distributed cache pattern:

- To optimize read-heavy workloads and improve system performance.
- For applications where data can be temporarily stale or slightly outdated without causing significant issues.

## Gossip Protocol:

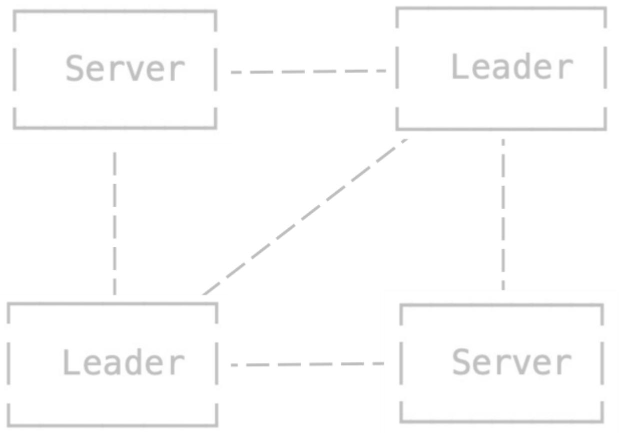


The gossip protocol is a method for spreading information across nodes in a network in a decentralized way. Nodes periodically exchange information with a few randomly chosen peers. Over time, information spreads through the network, achieving eventual consistency.

When to use the gossip protocol pattern:

- In systems with frequent node additions, departures, or changing network conditions.
- For scenarios where strong consistency isn't critical, and eventual consistency is acceptable.

## Random Leader Election:



In a random leader election pattern, nodes in a distributed system periodically participate in leader election processes. A leader is chosen randomly, and it's responsible for making decisions and coordinating actions among other nodes.

When to use the random leader election pattern:

- For scenarios where a single point of coordination (leader) is needed without relying on a fixed coordinator service.
- In systems where nodes are relatively homogeneous and capable of participating in leader election processes.

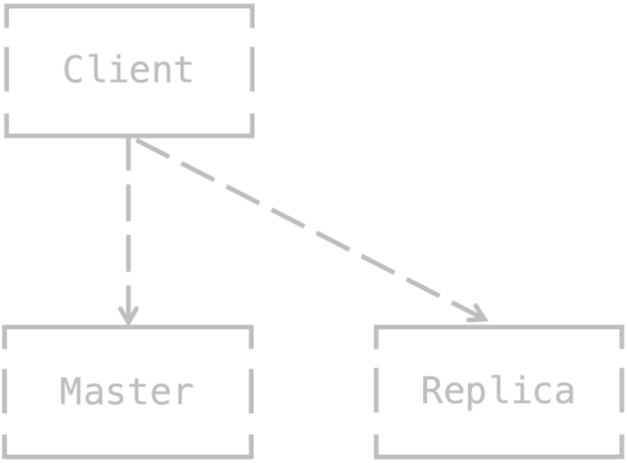
# AVAILABILITY

## Availability Patterns

Availability patterns are strategies used in system design to ensure that a system remains operational and accessible even in the face of failures or disruptions.

Here are two common availability patterns:

### Replication:

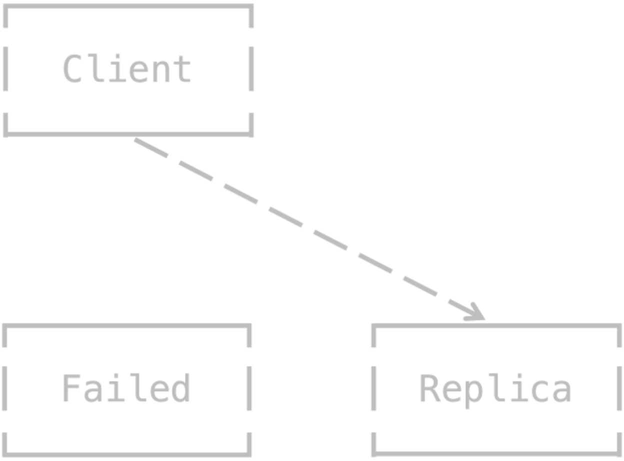


Replication involves creating duplicate copies of the system's components, services, or data on different servers or nodes. The primary goal of replication is to provide redundancy and fault tolerance. If one copy becomes unavailable due to a failure, another copy can take over, ensuring continuous availability.

When to use the replication pattern:

- Critical systems where downtime is unacceptable, such as financial platforms or emergency services.
- When you need to ensure high availability without manual intervention in case of failures.
- For read-heavy workloads, as replicas can offload read requests from the primary system.

### Failover:



Failover is a mechanism that automatically switches to a backup or secondary system when the primary system becomes unavailable.

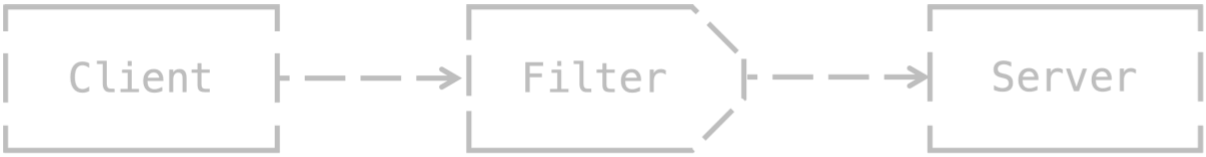
The backup system is kept up to date through replication. Failover ensures minimal downtime and seamless transitions when failures occur.

When to use the failover pattern:

- In scenarios where maintaining continuous service availability is a top priority.
- For systems where the impact of downtime, such as revenue loss or customer dissatisfaction, is significant.
- For systems that need to recover quickly and automatically from failures without manual intervention.

Both replication and failover patterns are used to improve system availability, but they tackle availability challenges from different angles.

## Throttling / Rate Limiting



Throttling or rate limiting is a technique used in system design to control the rate at which certain actions or requests are allowed to be performed by a user or a client. It helps prevent abuse, ensure fair usage of resources, and maintain system stability and availability by preventing overwhelming requests or excessive usage.

Throttling can be implemented in various ways, such as limiting the number of requests a user can make within a certain time period, delaying requests that exceed a certain rate, or temporarily blocking access for users who violate predefined limits.

How throttling helps with system **availability**:

## Prevent Overload:

Throttling prevents a single user or client from sending an excessive number of requests in a short period. Without throttling, a sudden influx of requests, whether intentional or due to a malfunction, could overwhelm the system's resources, causing slowdowns or outages.

## Mitigate DDoS Attacks:

Distributed Denial of Service (DDoS) attacks involve flooding a system with an overwhelming amount of traffic. Throttling helps mitigate the impact of such attacks by limiting the rate at which requests are processed, reducing the effectiveness of the attack.

## Fair Resource Allocation:

Throttling ensures that all users or clients get a fair share of the available resources. It prevents any single user from monopolizing resources, which can lead to a poor experience for other users.

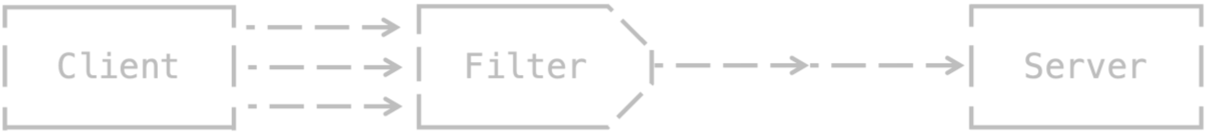
## Stabilize Performance:

By controlling the rate of incoming requests, throttling helps maintain stable and consistent performance. It prevents spikes in traffic that could strain resources and cause performance degradation.

## Prevent Service Degradation:

Throttling can be particularly useful during peak usage times. By limiting the rate of incoming requests, the system can continue to provide acceptable performance to all users without being overwhelmed.

## Queue-based Load Leveling



Queue-based Load Leveling is a technique used in system design to manage and distribute incoming requests or tasks in a way that evens out the load on the system, preventing sudden spikes in traffic from overwhelming the resources. This approach involves placing incoming requests in a queue and processing them at a controlled rate, ensuring that the system can handle the workload without degradation in performance or availability.

How Queue-based Load Leveling helps with system **availability**:

### Prevent Overloads:

During periods of high traffic or sudden bursts of requests, the system may become overloaded if it tries to process all requests simultaneously. Queue-based Load Leveling staggers the processing of requests, preventing spikes in usage that could lead to resource exhaustion or downtime.

### Mitigate Resource Contention:

When many requests compete for resources (such as CPU, memory, or network bandwidth), contention can arise, causing performance degradation. Load leveling spreads out the resource demands over time, reducing contention and maintaining stable performance.

### Enhance Scalability:

Queue-based Load Leveling enables the system to scale more effectively. Instead of provisioning resources to handle the highest possible load, the system can dynamically adjust to the current traffic while maintaining availability.

### Improve Fault Tolerance:

In distributed systems, load leveling can help manage the load across multiple nodes. If one node fails, the remaining nodes can continue processing the queued requests, preventing service disruptions.

### Smooth User Experience:

Load leveling helps maintain a consistent user experience by avoiding sudden slowdowns or timeouts during traffic spikes. Users are less likely to encounter performance degradation due to sudden resource shortages.

Implementing Queue-based Load Leveling involves creating a buffer (queue) that holds incoming requests and processes them in a controlled manner. Techniques like rate limiting, throttling, and dynamic scaling can be employed to adjust the rate at which requests are dequeued and processed based on the system's capacity and available resources.

## Health Checks

Health Endpoint Monitoring, also known as Health Checks, is a practice in system design and monitoring where the health and status of various components or services in a system are periodically checked by sending requests to designated health endpoints. The purpose of health endpoint monitoring is to ensure that the system's components are functioning correctly and to detect and address issues as early as possible.

How Health Endpoint Monitoring helps with system availability:

### Early Detection of Issues:

By continuously monitoring the health of system components, health checks can identify problems and anomalies early, allowing for timely investigation and resolution before they impact the overall system availability.

### Proactive Maintenance:

Health checks enable proactive maintenance. If a component is detected as unhealthy, appropriate actions can be taken to address the issue before it leads to service disruptions.

### Fault Isolation:

When a component fails or becomes unhealthy, health checks help pinpoint the exact component or service that is causing the problem. This makes it easier to isolate and address the issue, reducing downtime and impact on other parts of the system.

## Graceful Degradation:

Health checks can be used to implement graceful degradation strategies. If a certain service or component is found to be unhealthy, the system can automatically adjust its behavior to minimize the impact on users while the issue is being addressed.

## Load Balancer Decisions:

Health checks provide information to load balancers about the health and availability of backend servers. Load balancers can route traffic away from unhealthy servers, improving the overall availability of the system.

## Automatic Recovery:

When an unhealthy component recovers, health checks can trigger automatic processes to bring it back into the rotation, ensuring that the system fully recovers without manual intervention.

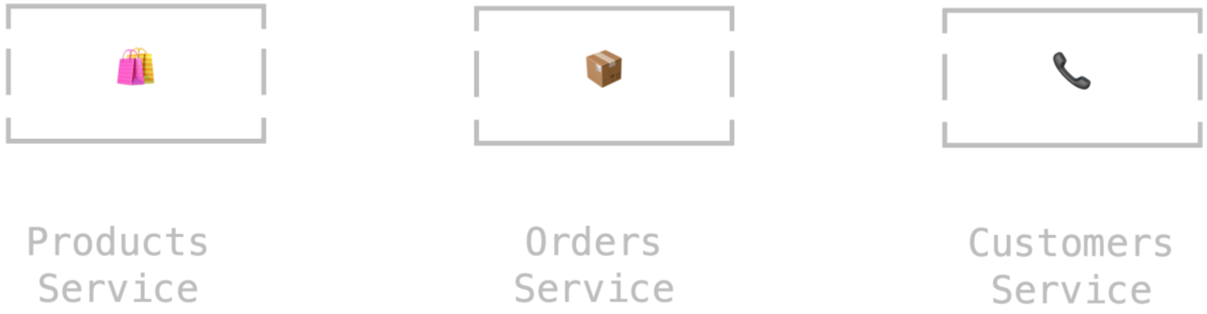
## Monitoring and Alerting:

Health checks are essential for monitoring and alerting systems. When a component's health status deviates from the expected, notifications can be sent to administrators or automated systems for immediate attention.

Implementing Health Endpoint Monitoring involves defining endpoints for each component or service in the system. These endpoints are typically lightweight and provide information about the component's health status, such as whether it's operational, responsive, and performing as expected.

# RESILIENCE

## Bulkhead



The Bulkhead Pattern is a fancy way to say that we can make our system extra tough by separating out different parts of it. It's like how ships have different compartments to keep water from flooding the whole thing if there's a leak.

Basically, we divide our system into different "bulkheads," each with its own job or group of users. That way, if one part fails, it won't take down the whole system. It's like having different backup plans to keep things running smoothly.

Key benefits of the Bulkhead Pattern for **resilience** include:

### Isolation of Failures:

If one part of the system experiences a failure, it won't cascade to other parts of the system, reducing the potential for widespread downtime or disruptions.

### Improved Availability:

Even if one section experiences issues, other sections can continue to serve users, enhancing the overall availability of the system.

### Easier Maintenance:

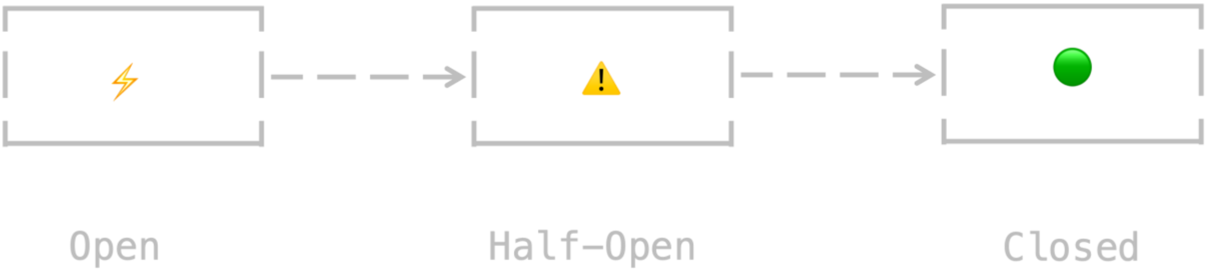
Isolating sections can simplify maintenance and updates, as changes in one section are less likely to affect others.

An example of applying the Bulkhead Pattern could be in a microservices architecture, where each microservice is encapsulated within its own bulkhead. If one microservice experiences a surge in traffic or a failure, it won't overload or bring down other microservices. Instead, they remain operational, providing a degree of fault tolerance.

Let's walk through a e-commerce example. Imagine you're designing an e-commerce application that consists of multiple backend services responsible for different functionalities: product management, order processing, and customer support.

The e-commerce application has three separate backend services. These separate services serve as bulkheads, isolating different functionalities.

## Circuit Breaker



The Circuit Breaker Pattern is a design pattern used in system architecture to enhance the resilience of a system by preventing repeated requests to a failing component or service. It is inspired by the concept of an electrical circuit breaker that stops the flow of electricity when there is an overload or fault to prevent damage.

In software, the Circuit Breaker Pattern serves as a mechanism to detect and manage failures, allowing a component to temporarily "open" the circuit and stop requests to a failing service. This prevents the system from overloading or becoming unresponsive due to continuous requests to a component that's already struggling.

Key characteristics of the Circuit Breaker Pattern and how it enhances resilience include:

### Failure Detection:

The Circuit Breaker continuously monitors the responses from a component or service. If a predefined threshold of failures or errors is reached, the Circuit Breaker transitions to a "broken" state.

## Fast Failures:

When the Circuit Breaker detects a broken state, it prevents further requests to the failing component. This helps the system to quickly fail-fast, reducing the load on the failing service.

## Fallback Mechanism:

In case of a broken circuit, the Circuit Breaker can provide a fallback mechanism. This can include using cached data, alternative services, or default responses to continue serving user requests without waiting for the failing service to recover.

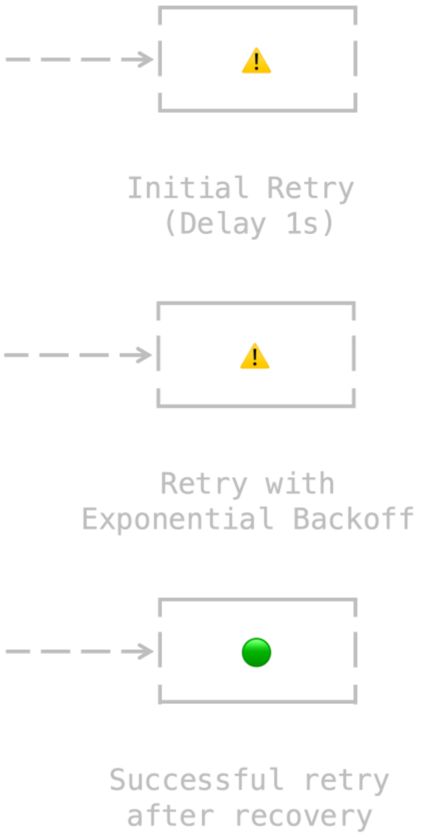
## Recovery and Half-Open State:

After a certain period, the Circuit Breaker can transition to a "half-open" state, allowing a limited number of test requests to the component. If these requests succeed, the circuit is closed again, and normal operation resumes. If they fail, the circuit remains open.

## System Resilience:

By preventing continuous requests to a failing component, the Circuit Breaker helps to protect other parts of the system from being affected by the failure.

# The Exponential Backoff and Retry



The Exponential Backoff and Retry Pattern is a design pattern used in system architecture to enhance the resilience of a system by handling transient failures that may occur during communication with external services or components.

This pattern is especially useful in distributed systems and when interacting with unreliable or intermittently available resources.

The pattern involves can include the following key concepts:

## Retry Mechanism:

When a request to an external service or component fails, the system automatically retries the operation after a brief delay.

## Exponential Backoff:

If the initial retry attempt fails, the system increases the delay before the next retry exponentially. This means that subsequent retries are spaced farther apart in time.

### Jitter:

To avoid synchronization and reduce the likelihood of multiple systems retrying simultaneously, a random "jitter" is often introduced to the backoff period.

### Maximum Retry Count:

The pattern includes a limit on the number of retries. If the operation continues to fail after a certain number of retries, the system may escalate the issue or take alternative actions.

This is how it enhances **resilience**:

### Transient Failure Handling:

Many system failures are transient and occur due to temporary network issues or resource unavailability. The pattern helps the system deal with these temporary failures without overloading resources.

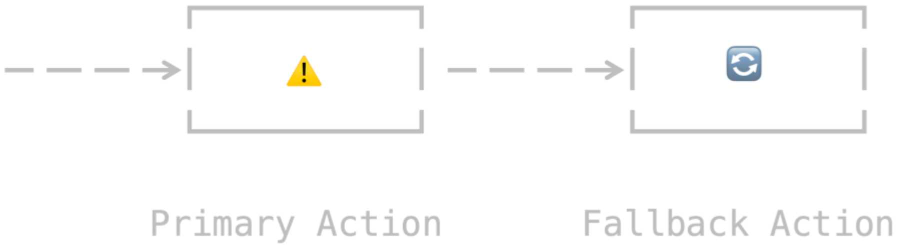
### Reduced Chances of Overload:

By introducing delays between retries, the system avoids bombarding a potentially overloaded service with a high volume of requests, which could lead to further failures.

### Improved User Experience:

Users experience less disruption when interacting with a system that employs exponential backoff and retry, as the pattern mitigates the impact of transient failures on user-facing operations.

# Fallback Mechanism



The Fallback Mechanism is a design approach used in system architecture to enhance the resilience of a system by providing an alternative course of action when a primary operation fails or encounters issues. This pattern is particularly valuable in distributed systems and scenarios where complex operations involve multiple steps that could potentially fail.

The key idea behind the Fallback Pattern is to have a predefined fallback mechanism or compensating transaction that can be executed if the primary operation cannot be completed successfully. This secondary operation aims to undo or compensate for the effects of the primary operation, ensuring that the system remains in a consistent state despite the failure.

How the Fallback Pattern enhances **resilience**:

## Graceful Degradation:

When a primary operation fails, the system can immediately trigger the fallback mechanism to handle the situation, preventing cascading failures and maintaining system functionality.

## Data Consistency:

Compensating transactions are designed to bring the system back to a consistent state, ensuring that any partial changes made by the primary operation are appropriately rolled back or compensated.

## Error Recovery:

The pattern offers a structured approach to handling errors, reducing the chances of leaving the system in an undefined state after a failure.

### Availability:

By providing an alternative way to achieve the same outcome, the system can remain available and operational even if the primary operation encounters issues.

### Predictable Behaviour:

Users and applications can rely on the fact that, regardless of whether the primary operation succeeds or fails, the system will always maintain its integrity through the use of compensating transactions.

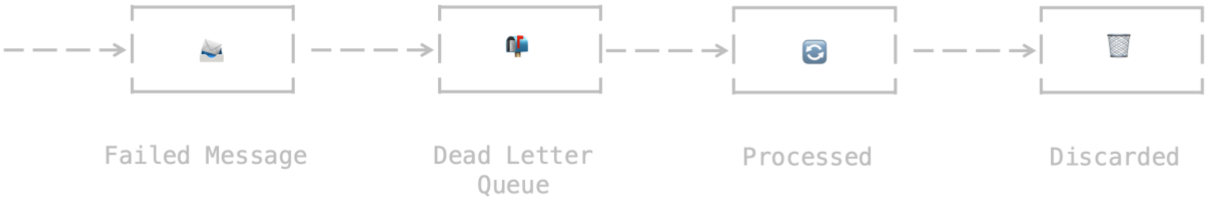
### Proper Error Handling:

For operations involving multiple steps, the Fallback Pattern provides a structured way to handle each step individually and ensure proper error handling.

Example use cases for the Fallback Pattern include e-commerce checkout processes, financial transactions, and any scenario where multiple steps are involved and failure of any step could potentially leave the system in an undesirable state.

The **Compensating Transaction Pattern** is a specialized form of the Fallback Pattern that specifically deals with complex transactions involving multiple steps. It emphasizes maintaining data integrity and consistency throughout the process, whereas the Fallback Pattern is more about offering an alternative way to achieve a goal when a primary operation fails.

## The Dead Letter Queue (DLQ)



The Dead Letter Queue (DLQ) is a design pattern commonly used in messaging systems to handle messages that cannot be successfully processed or delivered to their intended recipients. It's a mechanism that captures messages that encounter issues during processing, such as errors, malformed content, or unavailable recipients.

Instead of discarding these problematic messages, they are redirected to the DLQ for further analysis and potential resolution.

How the Dead Letter Queue pattern enhances resilience:

## Error Isolation:

Messages that encounter errors or failures don't disrupt the main processing flow. They are isolated in the DLQ, preventing the entire system from being affected by a single message's issues.

## Preservation of Data:

Instead of losing valuable data due to errors, the DLQ stores problematic messages for further investigation. This is especially useful for compliance, auditing, and debugging purposes.

## Guaranteed Delivery:

DLQs can be used to ensure that messages are not lost even in cases of temporary communication failures.

## Failed Process Recovery:

If a processing component fails, the messages it was working on can be recovered from the DLQ once the component is back online.

DLQs are especially valuable in scenarios with high message volumes, asynchronous communication, and complex workflows. They provide a safety net for the system by allowing it to continue processing without being blocked by occasional failures, and they provide a mechanism for addressing errors and improving system performance over time.

# SYSTEM DESIGN PATTERNS

## Bloom Filters

```
+---+---+---+---+---+---+---+---+---+---+

| 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 |

+---+---+---+---+---+---+---+---+---+---+
```

Bloom filters are a probabilistic data structure used in computer science and system design to quickly check whether an element is a member of a set. They provide an efficient way to perform membership queries with low memory usage.

### Concept of Bloom Filters:

A Bloom filter consists of an array of bits and a set of hash functions. When an element is added to the Bloom filter, its hash values are used to set the corresponding bits in the array. To check if an element is in the set, its hash values are hashed and checked against the bits in the array. If all the corresponding bits are set, it's likely that the element is in the set. However, if any of the bits are not set, the element is definitely not in the set.

### Usefulness of Bloom Filters:

Bloom filters are particularly useful in scenarios where memory is constrained, and false positives can be tolerated. Here are some common use cases where Bloom filters are applied:

Caching: Bloom filters can be used in caching systems to quickly determine if a requested item is present in the cache. This prevents unnecessary cache lookups for items that are not present, reducing load on the cache.

Distributed Systems: Bloom filters can help reduce the amount of network traffic in distributed systems. They can be used to determine which nodes might have certain data, avoiding unnecessary requests to nodes that definitely don't have it.

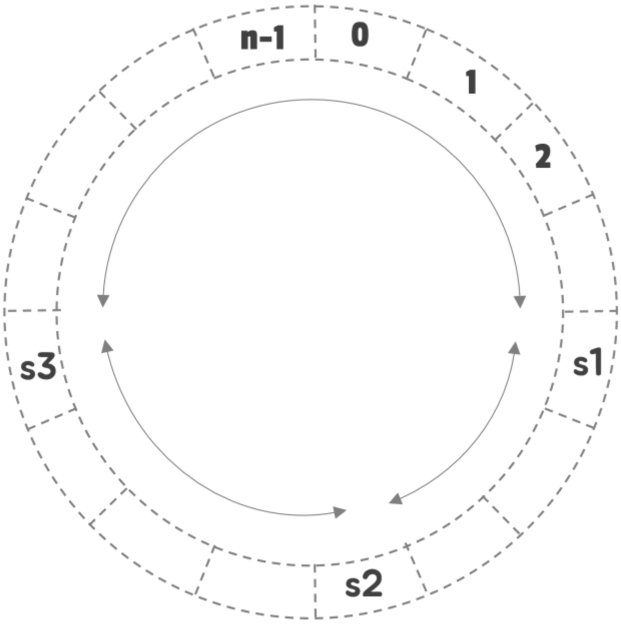
Spam Filtering: In email systems, Bloom filters can be employed to quickly identify whether an incoming email matches known spam patterns. This helps reduce the processing load for legitimate emails.

Big Data: Bloom filters can help optimize data processing pipelines by allowing quick filtering of data that is likely not relevant for a particular analysis or operation.

Web Crawling: In web crawling, Bloom filters can be used to prevent revisiting URLs that have already been crawled, saving bandwidth and time.

It's important to note that while Bloom filters provide fast membership queries, they can sometimes produce false positives (indicating an element is present when it's not). The probability of false positives can be controlled by adjusting the size of the filter and the number of hash functions used. The trade-off lies between memory usage and the acceptable level of false positives.

## Consistent Hashing



Consistent hashing is a technique used in distributed systems and caching to efficiently distribute data and load across multiple nodes or servers. It helps maintain balance, reduces data movement during node additions or removals, and allows for better scalability and fault tolerance.

### Concept of Consistent Hashing:

Imagine you have a distributed system with multiple nodes (servers) where data needs to be stored. Traditional hash functions might lead to data redistribution whenever nodes are added or removed. Consistent hashing addresses this issue by creating a ring-like structure (a virtual ring) where each node and data item are mapped.

Here's how it works:

Node Placement: Nodes are placed on the ring using a hash function. The hash values of nodes determine their positions on the ring.

Data Placement: Each data item is also hashed using the same hash function to find its position on the ring. The data item is placed on the node that comes after its position.

Finding Nodes: When a request for data comes in, the same hash function is applied to the data's key. The data is then placed on the node that comes after the hashed value on the ring.

## Usefulness of Consistent Hashing:

Consistent hashing offers several benefits in distributed systems:

Load Balancing: The distribution of data across nodes is more even, which prevents hotspots where some nodes are overloaded while others are underutilized.

Scalability: When nodes are added or removed, only a small portion of data needs to be remapped, minimizing data migration and disruption.

Fault Tolerance: If a node fails, its data can be quickly reassigned to the next available node without redistributing all data.

Caching: In caching systems, consistent hashing helps determine which cache node to use for storing or retrieving data.

Distributed Hash Tables: Consistent hashing forms the basis for distributed hash tables (DHTs), which allow efficient decentralized data storage and retrieval.

Reduced Churn: With traditional hash functions, node additions/removals cause massive data migration. Consistent hashing minimizes this churn.

Overall, consistent hashing is a powerful technique that improves the efficiency, scalability, and fault tolerance of distributed systems by providing a stable way to map data and nodes while minimizing disruption during changes.

## Checksum

```
    +---+---+---+---+

Data| 1 | 2 | 3 | 4 |

    +---+---+---+---+

Checksum: 10
```

In the context of distributed systems, a checksum is a simple but important concept used to ensure the integrity of data during transmission or storage. It's a small piece of data calculated from the original data to detect errors, corruption, or changes that might have occurred. Checksums are used to verify the accuracy of data and to provide confidence that the data hasn't been tampered with.

### Concept of Checksum:

A checksum is like a unique fingerprint for a piece of data. When data is sent or stored, a checksum value is calculated using a specific algorithm, often involving mathematical operations. This checksum value is then sent along with the actual data.

When the data reaches its destination or is retrieved from storage, the recipient calculates the checksum of the received data using the same algorithm. If the calculated checksum matches the received checksum, it indicates that the data is likely intact and hasn't been corrupted during transmission or storage.

### Usefulness of Checksums:

Checksums are extremely useful in distributed systems for several reasons:

Error Detection: Checksums can detect errors caused by transmission issues, hardware faults, or other forms of data corruption. If the calculated checksum doesn't match the received checksum, it signals that the data may have been compromised.

Data Integrity: In distributed databases, files, or communication protocols, checksums help maintain the integrity of the data. They provide confidence that the data hasn't been altered without authorization.

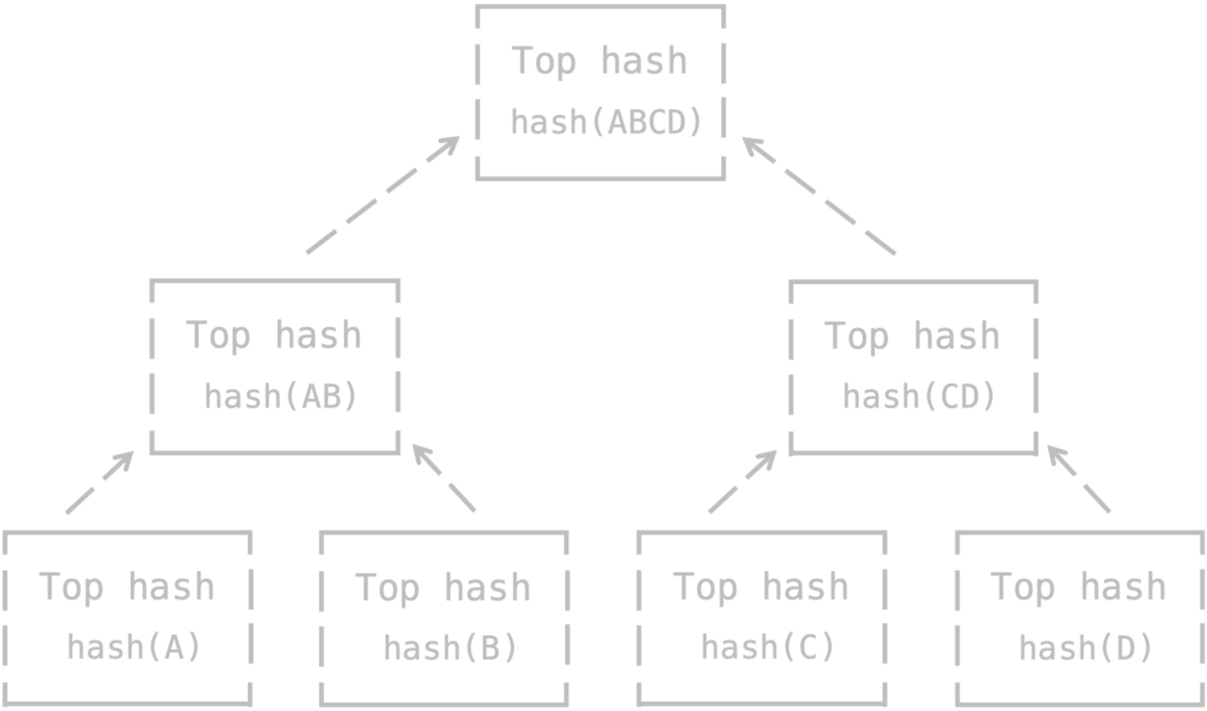
Data Verification: Checksums help validate that the data transferred between different components of a distributed system is accurate and complete. This is particularly crucial in critical applications where data accuracy is paramount.

Data Recovery: In case of data corruption, checksums can help identify which portions of the data are affected, aiding in recovery efforts.

Network Transmission: In networking, checksums are often used in packet headers to verify the integrity of the data being sent across the network.

Overall, checksums play a vital role in maintaining data reliability and integrity within distributed systems. They're a fundamental tool to ensure that the data being processed, transmitted, or stored remains accurate and unaltered, even in the face of potential errors or disruptions.

## Merkle Trees



In the context of system design, particularly in distributed systems and cryptography, a Merkle Tree (also known as a hash tree) is a data structure used to efficiently verify the integrity of large datasets, detect changes in data, and enable secure communication between multiple parties. Merkle Trees are widely used in blockchain technology, distributed databases, and digital signatures.

## Concept of Merkle Trees:

A Merkle Tree is a hierarchical structure composed of hash values. It works by recursively hashing pairs of data until a single root hash is obtained. Here's how it works:

Leaf Nodes: The original data is divided into fixed-size chunks or blocks. Each block is hashed individually to create leaf nodes of the tree.

Intermediate Nodes: The leaf nodes are then paired and hashed together to create intermediate nodes. This process continues, pairing and hashing nodes at each level, until only a single root hash remains.

Root Hash: The root hash is a condensed representation of the entire dataset. Any change in the data, no matter how small, will result in a completely different root hash.

## Usefulness of Merkle Trees:

Merkle Trees provide several benefits in distributed systems and cryptography:

Data Integrity Verification: By comparing root hashes, parties can quickly verify whether a large dataset has been tampered with or modified.

Efficient Data Comparison: Merkle Trees allow efficient verification of specific portions of data. This is useful for quickly identifying which parts of a large dataset have changed.

Blockchain: In blockchain technology, Merkle Trees enable efficient verification of transactions. The root hash of a Merkle Tree is stored in the blockchain, providing a compact and tamper-evident representation of all transactions.

Distributed Databases: In distributed databases, Merkle Trees are used to synchronize data between nodes and ensure consistency. Nodes can compare root hashes to detect inconsistencies.

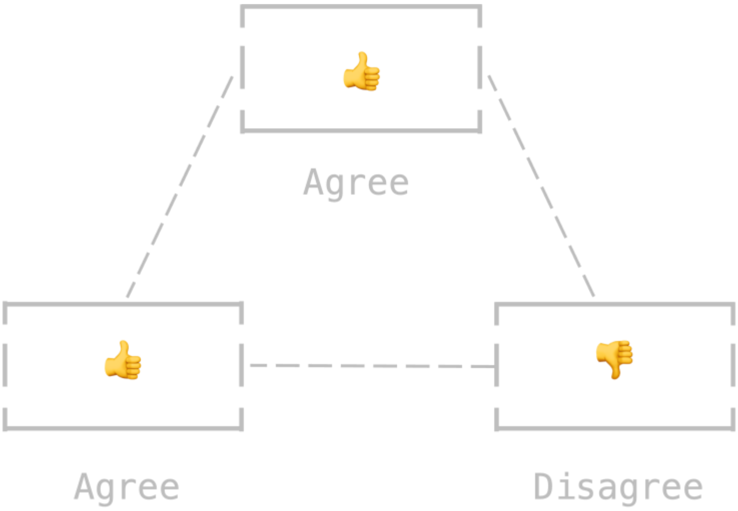
Peer-to-Peer Networks: Merkle Trees enable efficient validation of data shared between peers, enhancing the security and trustworthiness of data exchange.

Digital Signatures: Merkle Trees are used to create digital signatures that prove the authenticity of a document without revealing its content.

Efficient Proofs: Merkle Trees allow the creation of succinct proofs that a specific piece of data is included in a larger dataset without needing to share the entire dataset.

Overall, Merkle Trees are a powerful tool for maintaining data integrity, facilitating efficient data verification, and enabling secure communication in various distributed systems. They provide a way to efficiently prove the authenticity of data while minimizing the need to transmit or store large amounts of information.

## Quorum



A quorum represents a threshold that needs to be met to ensure that a distributed system operates effectively and reliably. Quorums are often used in scenarios where multiple nodes or components need to agree on a decision before proceeding. Different types of quorums exist, such as read quorums and write quorums, each serving different purposes.

For example, in a distributed database or storage system, a read quorum might require that data is read from a minimum number of nodes before returning a result to ensure consistency. Similarly, a write quorum might require that data is written to a certain number of nodes before considering a write operation successful.

## Usefulness of Quorums:

Quorums provide several benefits in distributed systems:

Consistency: Quorums help maintain data consistency by ensuring that a sufficient number of nodes agree on the state of the system. This prevents situations where nodes have conflicting views of the data.

Availability: Quorums enable systems to continue operating even if some nodes are unavailable. As long as the required number of nodes are operational, decisions can still be made.

Fault Tolerance: Quorums improve fault tolerance by allowing the system to tolerate failures of a certain number of nodes while still maintaining operational integrity.

Preventing Split-Brain: In scenarios where network partitions might occur, quorums can help prevent a "split-brain" situation where nodes on different sides of the partition continue to operate independently.

Decentralized Decision-Making: Quorums allow for decentralized decision-making in systems where nodes have equal authority. Decisions can be made collaboratively without relying on a central point of control.

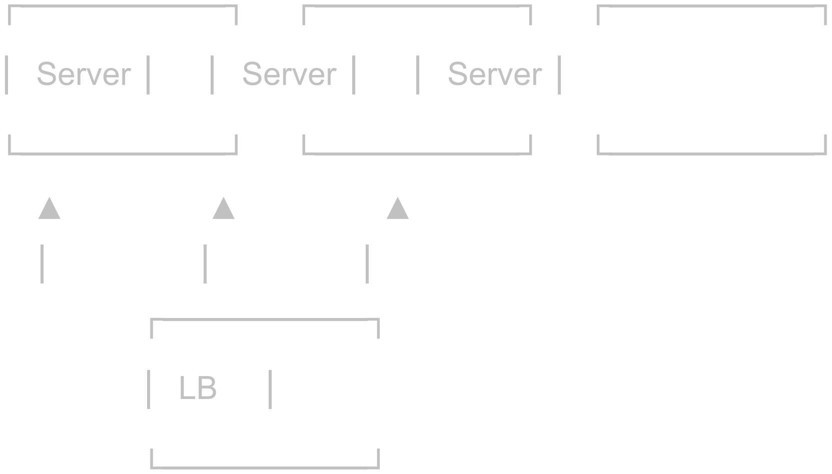
Scalability: Quorums can be adjusted based on the desired level of consistency, availability, and fault tolerance. This allows systems to scale while maintaining desired properties.

# LOAD-BALANCER

## Load Balancer vs Reverse Proxy

Load Balancer and Reverse Proxy are both components used in system design to optimize the performance, availability, and security of web applications. However, they serve different primary purposes and have distinct functionalities:

### Load Balancer:



A Load Balancer is a networking device or software that distributes incoming network traffic across multiple servers or resources. Its primary goal is to evenly distribute the incoming workload to ensure efficient resource utilization, prevent server overload, and enhance application responsiveness. Load balancers can be hardware-based appliances or software-based solutions.

### Key Functions of Load Balancers:

**Traffic Distribution**: Load balancers distribute incoming requests (HTTP, TCP, etc.) among multiple backend servers based on defined algorithms (round-robin, least connections, etc.).

**Scalability**: Load balancers support horizontal scaling by adding more servers to the backend pool, allowing the system to handle increased traffic.

**High Availability**: Load balancers ensure that if a server fails, traffic is automatically rerouted to healthy servers, minimizing downtime.

**Health Monitoring**: Load balancers regularly check the health of backend servers, removing or redirecting traffic from unhealthy servers.

## Reverse Proxy:

```
┌──────────┐
│ Client  │
└──────────┘
   ▲   ▲
   │   │
┌──────────┐
│ Reverse │
│ Proxy   │
└──────────┘
   ▼   ▼
┌──────────┐
│  Server │
└──────────┘
```

A Reverse Proxy, also known as an Application Delivery Controller (ADC), is a server or software component that sits in front of backend servers and acts as an intermediary between client requests and the backend servers. Its primary purpose is to provide additional layers of security, content caching, and application optimization.

### Key Functions of Reverse Proxies:

Security: Reverse proxies shield backend servers from direct client access, providing an extra layer of security by filtering out malicious traffic and protecting sensitive data.

Caching: Reverse proxies can cache static content, such as images and files, reducing server load and improving response times for frequently accessed resources.

SSL Termination: Reverse proxies can handle SSL encryption and decryption, offloading this resource-intensive task from backend servers.

Content Compression: Reverse proxies can compress content before sending it to clients, reducing bandwidth usage and improving page load times.

URL Routing and Rewriting: Reverse proxies can route requests based on URL patterns and rewrite URLs for cleaner, user-friendly paths.

### Comparison:

In summary, while both Load Balancers and Reverse Proxies contribute to better system performance and security, they focus on different aspects of optimization:

- Load Balancer: Focuses on distributing incoming traffic to ensure even resource utilization and improved availability of backend servers.
- Reverse Proxy: Focuses on enhancing security, content delivery, and application optimization through caching, SSL termination, and content compression.

In many real-world scenarios, Load Balancers and Reverse Proxies are used together, with the Load Balancer distributing traffic among multiple backend servers, and the Reverse Proxy providing additional security, caching, and optimization features.

## Load Balancing Algorithms

There are several popular load balancing algorithms that are commonly used to distribute incoming traffic among backend servers or resources in a load balancer. Each algorithm has its own characteristics and is suited for different scenarios. Some of the most popular load balancing algorithms include:

Round Robin: This algorithm distributes requests sequentially to each backend server in a circular order. It's simple to implement and provides relatively even distribution, but it doesn't consider the server's load or capacity.

**Least Connections**: Servers with the fewest active connections receive new requests. This algorithm helps to distribute traffic more evenly based on the current load of each server.

**Weighted Round Robin**: Similar to Round Robin, but servers are assigned different weights based on their capacity. Servers with higher weights receive more traffic, which is useful for balancing servers with varying capabilities.

**Weighted Least Connections**: Similar to Least Connections, but servers are assigned different weights based on their capacity. Servers with higher weights receive fewer connections, distributing traffic according to capacity.

**IP Hash**: The IP address of the client is used to determine which server to send the request to. This is useful for ensuring that requests from the same client consistently go to the same server.

**Least Response Time**: Servers with the lowest response time to the client receive new requests. This algorithm aims to direct traffic to the server that can respond most quickly.

**Adaptive Load Balancing**: This algorithm dynamically adjusts server weights based on real-time performance metrics, allowing the load balancer to adapt to changing server conditions.

Different load balancing algorithms have their strengths and weaknesses. The choice of algorithm depends on factors such as the architecture of the application, the characteristics of the backend servers, and the desired behavior in terms of traffic distribution and server load.

# Layer 4 Load Balancing vs Layer 7 Load Balancing

```
+---------------------------+
|        Application        |
+---------------------------+
|        Presentation        |
+---------------------------+
|          Session           |
+---------------------------+
|          Transport         |
+---------------------------+
|          Network           |
+---------------------------+
|           Data Link        |
+---------------------------+
|           Physical         |
+---------------------------+
```

Layer 4 Load Balancer and Layer 7 Load Balancer are two types of load balancers used in system design to distribute incoming network traffic to backend servers. They operate at different layers of the network stack and offer distinct functionalities:

## Layer 4 Load Balancer:

A Layer 4 Load Balancer operates at the transport layer (Layer 4) of the OSI model. It primarily makes routing decisions based on information available in the transport layer headers, such as source IP address, destination IP address, source port, and destination port. This type of load balancer is more focused on distributing traffic based on network-level information without deep inspection of application content.

## Key Characteristics of Layer 4 Load Balancer:

Traffic Distribution: Layer 4 load balancers distribute incoming requests to backend servers based on IP addresses and port numbers.

Network-Level Load Balancing: They are well-suited for scenarios where the backend servers host similar applications or services that can be identified by IP and port.

Fast Routing Decisions: Layer 4 load balancers make routing decisions quickly because they don't analyze application-level content.

Connection Handling: They handle network connections efficiently, ensuring even distribution of traffic.

### Layer 7 Load Balancer:

A Layer 7 Load Balancer operates at the application layer (Layer 7) of the OSI model. It can make routing decisions based on more sophisticated criteria, such as specific URLs, HTTP headers, cookies, or application-specific data. Layer 7 load balancers have the capability to inspect application-level content and can therefore make more granular and context-aware routing decisions.

### Key Characteristics of Layer 7 Load Balancer:

Application-Level Load Balancing: Layer 7 load balancers distribute traffic based on application-specific information, enabling advanced routing decisions.

Content Inspection: They can inspect HTTP headers, cookies, and other application-level data to determine the appropriate backend server.

Load Balancing Strategies: Layer 7 load balancers can employ more complex load balancing algorithms that take into account server health, response times, and user sessions.

Request-Based Routing: They can route requests to different backend servers based on the requested content, enabling optimizations such as caching and content delivery.

In summary, the primary difference between Layer 4 and Layer 7 Load Balancers lies in the layer of the network stack at which they operate and the type of information they use to make routing decisions. Layer 4 Load Balancers focus on network-level information, while Layer 7 Load Balancers analyze application-level content for more sophisticated and context-aware routing decisions. The choice between the two depends on the specific requirements of the application and the desired level of traffic distribution granularity.

## Domain Name System (DNS)

```
           DNS Root Servers
                        |
         +--------------|--------------+
         |              |              |
 .com TLD  .org TLD  .net TLD  ... Other TLDs
         |              |              |
    +----+----+    +----+----+    +----+----+
    |         |    |         |    |         |
google    amazon   wikipedia   ...   (Other Domains)
```

The Domain Name System (DNS) is a critical component of the internet infrastructure that translates human-readable domain names (like www.example.com) into IP addresses (such as 192.0.2.1) that computers use to identify and communicate with each other. DNS serves as a distributed, hierarchical, and highly available naming system, providing a way to locate resources on the internet.

## Key Functions and Benefits of DNS:

Name Resolution: DNS allows users to access websites and services using easily memorable domain names instead of numeric IP addresses. It simplifies the process of accessing online resources.

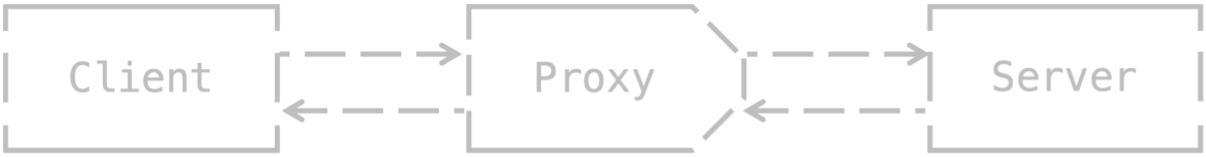
Hierarchical Structure: DNS employs a hierarchical structure with multiple levels of domain names, such as top-level domains (TLDs) like .com, .org, and country-code TLDs like .uk. This structure enables efficient and organized name resolution.

Load Distribution: DNS can be used for load distribution and balancing by distributing incoming traffic among multiple IP addresses associated with a single domain name.

Reverse DNS Lookup: DNS also supports reverse lookup, which allows you to find the domain name associated with a given IP address. This is useful for network diagnostics and security purposes.

Dynamic IP Assignment: DNS can be used to associate dynamic IP addresses with domain names, allowing resources like websites or applications to be accessible even if their IP addresses change periodically.

## Proxy (Forward proxy)



In the context of system design, a proxy, specifically a forward proxy, is an intermediate server that sits between clients and the servers they want to access. It acts as an intermediary, forwarding requests from clients to servers and returning responses from servers to clients. Forward proxies are often used to improve privacy, security, and performance for clients accessing resources on the internet.

### Key Functions and Benefits of Forward Proxies:

Privacy and Anonymity: Clients can use a forward proxy to access resources on the internet without revealing their own IP addresses. This can help protect user privacy and anonymity.

Content Filtering and Access Control: Forward proxies can enforce content filtering policies by blocking or allowing access to specific websites or content categories. They can also restrict access to certain domains or URLs, improving security and compliance.

Bandwidth Savings: By caching and serving resources locally, forward proxies can reduce the amount of data transferred between clients and servers, leading to reduced bandwidth usage and faster loading times.

Access to Restricted Content: Forward proxies can be used to access resources that might be geographically restricted, allowing clients to bypass content restrictions imposed by websites.

Network Optimization: Forward proxies can optimize network traffic by compressing data, removing unwanted headers, and reducing the overall data transferred between clients and servers.

Security and Anomaly Detection: Forward proxies can analyze incoming and outgoing traffic for security threats, helping to detect and prevent malicious activities. They can also log and analyze network traffic patterns for anomaly detection.

Authentication and Single Sign-On: Forward proxies can enforce user authentication, requiring users to log in before accessing certain resources. This is particularly useful for controlling access to internal corporate resources.

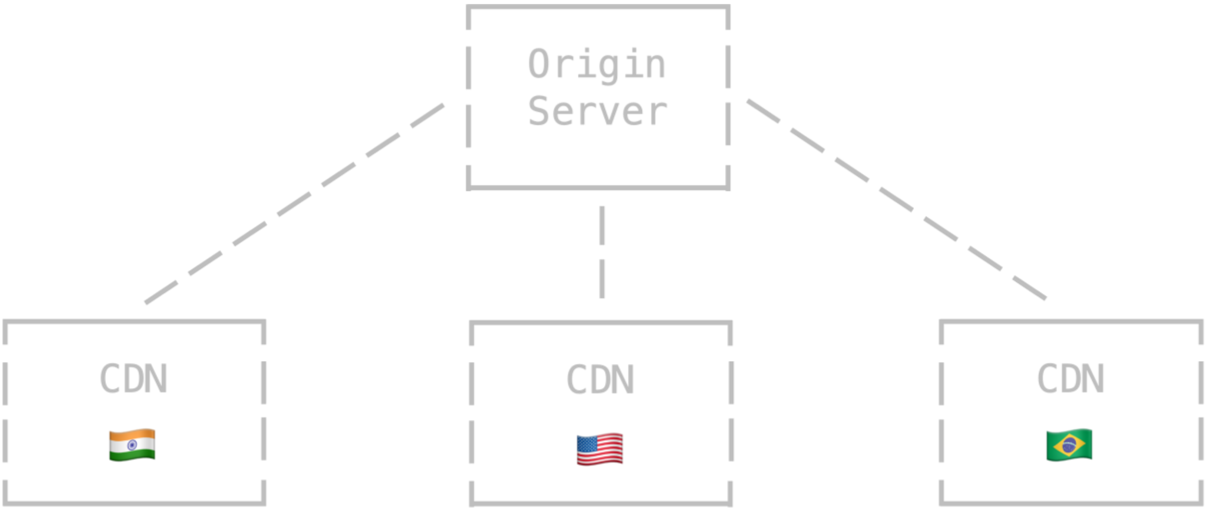
Bypassing Firewalls: In certain scenarios, forward proxies can be used to bypass network firewalls or access resources that might be blocked by network policies.

In summary, a forward proxy is a versatile component in system design that serves as an intermediary between clients and servers. It offers various benefits, including privacy, security, caching, and network optimization. By sitting between clients and servers, forward proxies can enhance the overall user experience, improve resource access times, and provide additional layers of control and protection for both clients and the systems they interact with.

### Forward vs Reverse Proxy

The difference between a forward and reverse proxy is subtle but important. A simplified way to sum it up would be to say that a forward proxy sits in front of a client and ensures that no origin server ever communicates directly with that specific client. On the other hand, a reverse proxy sits in front of an origin server and ensures that no client ever communicates directly with that origin server.

## Content Delivery Network (CDN)



Content Delivery Network (CDN) is a distributed network of servers strategically placed across various geographic locations.

The primary purpose of a CDN is to improve the delivery of web content, such as images, videos, stylesheets, and scripts, to users by reducing latency, increasing availability, and optimizing content delivery. CDNs work by caching and delivering content from the server location that is closest to the user's geographic location.

## Key Functions and Benefits of CDNs:

Latency Reduction: CDNs reduce the distance between the user and the server, which minimizes the latency and improves the loading speed of web content.

Content Caching: CDNs cache copies of content in various server locations. When a user requests a resource, the CDN serves the cached copy, reducing the load on the origin server and improving response times.

Load Distribution: CDNs distribute incoming traffic across multiple server locations, preventing any single server from becoming overloaded. This enhances the scalability and reliability of the website or application.

Distributed Network: CDNs have servers in multiple regions and data centers, providing redundancy and increased availability. This is particularly useful in mitigating the impact of server failures or network outages.

Global Reach: CDNs enable websites and applications to be accessible and performant for users around the world, regardless of their geographic location.

Traffic Offloading: CDNs offload a significant portion of traffic from the origin server, allowing the origin server to focus on dynamic content generation and processing.

## Pull CDN vs. Push CDN:

The main difference between Pull and Push CDNs lies in how content is placed on the CDN servers:

## Pull CDN:

- In a Pull CDN, the CDN servers retrieve content from the origin server on-demand when a user requests that content.
- When a user requests a resource, the CDN server fetches the content from the origin server, caches it, and serves it to the user.
- Benefits include reduced load on the origin server and efficient utilization of storage on the CDN servers.

## Push CDN:

- In a Push CDN, content is manually uploaded or pushed to the CDN servers in advance of user requests.
- The content is uploaded to the CDN servers' cache proactively, regardless of whether users have requested it yet.
- Push CDNs are suitable for static content that doesn't change frequently.
- Benefits include immediate content availability and control over when content is distributed to the CDN.

In summary, CDNs play a crucial role in system design by optimizing content delivery, reducing latency, improving availability, and enhancing the overall user experience. They can significantly improve the performance and scalability of websites and applications by leveraging a distributed network of servers strategically located across the globe. The choice between Pull and Push CDNs depends on the nature of the content, update frequency, and the desired level of control over content distribution.

# APPLICATION LAYER

## N-tier applications

```
+----------------------+

|                      |

|   Presentation Layer   |

|                      |

+----------------------+

|                      |

|   Application Layer   |

|                      |

+----------------------+

|                      |

|   Business Logic Layer |

|                      |

+----------------------+

|                      |

|   Data Access Layer    |

|                      |
+-------------------------+

|                    |

|   Integration Layer     |

|                    |

+-------------------------+
```

In system design, N-tier applications refer to the architectural pattern of designing software applications with multiple distinct layers or tiers, each responsible for specific functions and tasks. This architectural approach is used to separate concerns, improve modularity, enhance maintainability, and facilitate scalability in complex applications.

## Key Tiers in N-tier Applications:

1. **Presentation Tier (UI Tier):**
   - This is the topmost layer that interacts directly with users.
   - It handles user interfaces, user interactions, and presentation logic.
   - Typically includes web browsers, mobile apps, or desktop clients.
2. **Application Logic Tier (Business Logic Tier):**
   - This layer contains the business logic and rules of the application.
   - It processes user input, performs calculations, and orchestrates the interactions between different components.
   - Often referred to as the "brains" of the application.
3. **Data Access Tier (Data Tier or Persistence Tier):**
   - This layer manages the storage and retrieval of data.
   - It interacts with databases, file systems, and external APIs to store and retrieve data.
   - Responsible for ensuring data integrity and security.
4. **Integration Tier (Services Tier or Communication Tier):**
   - This layer handles communication and integration between different parts of the application or with external services.
   - It may expose APIs or services that other tiers can consume.

- Ensures data flow and interaction between components.

## Benefits of N-tier Applications:

Modularity: The separation of concerns into distinct tiers makes the application more modular and easier to develop, test, and maintain. Changes in one tier have minimal impact on others.

Scalability: Different tiers can be scaled independently to handle varying levels of load. For instance, the data access tier can be optimized for database queries, and the application logic tier for processing business rules.

Reusability: By encapsulating specific functionalities within tiers, components can be reused across different parts of the application or even in other applications.

Maintenance: Changes, updates, and enhancements can be made more easily due to the clear separation of functionality into distinct tiers.

## Example:

Let's say a user visits an online shopping website to purchase items. Here's how the n-tier architecture would be applied:

**Presentation Tier (Client):** The user opens a web browser and interacts with the website's user interface. They browse through products, view product details, and add items to their shopping cart.

**Application Tier (Business Logic):** The application tier receives requests from the presentation tier. It processes user actions, such as adding items to the cart, calculating the total price, and validating user inputs. It communicates with the services tier for tasks like authenticating users and processing payments.

**Services Tier:** The services tier provides authentication services to verify user credentials. It also handles payment processing by interfacing with external payment gateways.

**Data Tier (Database):** The data tier stores information about products, user accounts, and orders. It retrieves product details, updates cart contents, and records order history.

## Microservices

```
+---------------------+
|   Microservice A    |
|   +-----------+     |
|   |           |     |
|   |   Func A  |     |
|   |           |     |
|   +-----------+     |
|                     |
+---------------------+

+---------------------+
|   Microservice B    |
|   +-----------+     |
|   |           |     |
|   |   Func B  |     |
|   |           |     |
|   +-----------+     |
|                     |
+---------------------+

+---------------------+
|   Microservice C    |
|   +-----------+     |
|   |           |     |
|   |   Func C  |     |
|   |           |     |
|   +-----------+     |
|                     |
+------------------------+
```

In system design, Microservices is an architectural style and approach where a software application is decomposed into small, independent, and loosely coupled services that communicate with each other through well-defined APIs.

Each microservice is responsible for a specific business capability or function, and they can be developed, deployed, and scaled independently. This approach contrasts with monolithic architectures where the entire application is tightly integrated into a single codebase.

## Key Characteristics of Microservices:

Decomposition: The application is divided into smaller, manageable components called microservices, each focusing on a specific business domain or functionality.

Independence: Each microservice is developed, deployed, and managed independently. This enables teams to work on different services simultaneously without interfering with each other's work.

Loose Coupling: Microservices communicate through well-defined APIs and protocols, allowing them to evolve and change independently without affecting other services.

Technology Diversity: Different microservices can be developed using different technologies and programming languages that best suit their specific requirements.

Scalability: Microservices can be scaled individually based on demand, allowing for efficient resource utilization.

Resilience and Fault Isolation: Failures in one microservice do not necessarily impact the entire application. Services can be designed to handle failures and degrade gracefully.

Polyglot Persistence: Each microservice can use its own data storage technology, allowing for the best fit for different data storage requirements.

## Challenges:

Complexity: Managing multiple microservices and their interactions can introduce complexity, especially in larger systems.

Service Discovery: As the number of microservices grows, managing service discovery and communication can become challenging.

Data Consistency: Maintaining data consistency across multiple microservices can be complex, particularly in scenarios involving transactions.

Operational Overhead: The deployment, monitoring, and management of numerous microservices can create operational challenges.

# Service Discovery

```
+----------------------+
|   Service Registry   |
|                      |
|  +------+  +------+  |
|  | Svc1 |  | Svc2 |  |
|  |      |  |      |  |
|  |      |  |      |  |
|  +------+  +------+  |
|                      |
+----------------------+
         |
         | Discover
         |
+----------------------+
|       Service        |
|       Consumer       |
|                      |
+----------------------+
```

In system design, Service Discovery is a mechanism used in distributed architectures, particularly in microservices environments, to automatically locate and keep track of available services within a network.

It helps applications and services discover the network addresses and locations of other services they need to interact with.

Service Discovery is essential for ensuring seamless communication and collaboration between various components in a dynamic and rapidly changing environment.

Some of the most popular systems for service discovery are: etcd, Consul, ZooKeeper, Kubernetes Native Service Discovery: etc.

## How Service Discovery Works:

1. **Registration:**
    - Service registers itself with Service Discovery.
    - Provides information like name, address, port, and health status.
2. **Heartbeats and Health Checks:**
    - Service periodically sends heartbeats to indicate its health.
    - If no heartbeats or unhealthy status, service is marked as unavailable.
3. **Querying:**
    - Service queries Service Discovery to find another service's location.
    - Provides service name/identifier.
4. **Load Balancing:**
    - Service Discovery can return multiple instances for load balancing.
5. **Updating and Deregistration:**
    - Services continuously update registration status.
    - Deregister when service stops or is removed.

## Benefits of Service Discovery:

Dynamic Environment: In highly dynamic environments where services come and go frequently, Service Discovery provides a central mechanism to track service availability.

Decoupling: Services can interact without needing to know the specific network addresses or locations of each other, promoting loose coupling.

Scalability: As the number of services increases, Service Discovery ensures that services can efficiently discover each other, avoiding the need for manual configuration.

Resilience: In the face of service failures or outages, Service Discovery can help reroute traffic to healthy instances of the service.

Easy Deployment: New services can be deployed without requiring manual updates to the configurations of other services that need to communicate with them.

Load Balancing: By providing information about multiple instances of a service, Service Discovery enables load balancers to distribute traffic effectively.

In summary, Service Discovery is a vital component in modern distributed architectures, helping services dynamically discover and communicate with each other without manual intervention. It enables system flexibility, scalability, and resilience while addressing challenges related to service management and communication in dynamic environments.

## API Gateway

```
+----------------------+
|    API Gateway       |
|                  |
| +-----------+      |
| |  Service  |      |
| |  Routing  |      |
| | and Proxy |      |
| |  Logic    |      |
| +-----------+      |
|                  |
+----------------------+
   |     |     |
   |     |     |
+---|-------|-------|---+
|  |      |      |  |
|  |      |      |  |
| +---+   +---+   +---+ |
| | S1|   | S2|   | S3| |
| |   |   |   |   |   | |
| +---+   +---+   +---+ |
|                       |
+-----------------------+
```

In system design, an API Gateway is a server or service that acts as an entry point for client requests to a microservices architecture or distributed system. It provides a centralized point of control and management for APIs (Application Programming Interfaces) offered by various services.

The API Gateway handles tasks such as routing, authentication, security, load balancing, and protocol translation, streamlining the communication between clients and the underlying services.

## Key Concepts of API Gateway:

Centralized Entry Point: The API Gateway serves as a single entry point for client applications to access various APIs provided by different microservices.

Routing and Load Balancing: The API Gateway routes incoming requests to the appropriate microservices based on the requested endpoint. It can also distribute incoming traffic across multiple instances of the same microservice to ensure load balancing.

Authentication and Authorization: The API Gateway handles user authentication and authorization, enforcing security policies and ensuring only authorized users access the APIs.

Protocol Translation: The API Gateway can translate requests and responses between different protocols and formats, allowing clients and services to communicate in their preferred formats.

Aggregation: The API Gateway can aggregate data from multiple services to fulfill a single client request, reducing the number of calls the client needs to make.

## Advantages of API Gateway

Simplified Client Interaction: Clients interact with a single API Gateway endpoint, reducing the complexity of managing multiple service endpoints.

Security and Authentication: The API Gateway centralizes security measures, ensuring consistent authentication and authorization across all services.

Protocol Transformation: The Gateway can translate between protocols, enabling communication between clients and services that use different formats.

## Use Cases of API Gateway:

Microservices Architecture: An API Gateway is often used in microservices architectures to provide a unified API entry point for clients.

Mobile and Web Applications: API Gateways simplify the interaction between mobile apps, web clients, and backend services.

Third-Party Integrations: API Gateways can manage third-party API integrations, abstracting the complexities of working with external APIs.

## Example:

Consider an e-commerce platform with various microservices for product catalog, user management, and order processing.

The API Gateway receives incoming requests from clients, such as mobile apps and web browsers. It authenticates users, routes requests to the appropriate microservices (e.g., retrieving product information or placing orders), and aggregates data if needed. The API Gateway also handles rate limiting and caching to improve performance.

## Event Driven Architecture (EDA)

```
+------------------------------+
|                              |
|       Event Producer         |
|                              |
+------------------------------+
            |
            | Publishes Events
            |
            v
+------------------------------+
|                              |
|       Event Broker           |
|                              |
+------------------------------+
            |
            | Routes Events
            |
            v
+------------------------------+
|                              |
|       Event Consumers        |
|                           |

+----------------------------------+
```

In system design, Event-Driven Architecture (EDA) is an approach that focuses on the communication and interaction between different components or services based on events.

An event is a significant occurrence or state change that can be detected and processed by the system. EDA emphasizes the propagation of events and the corresponding reactions or actions triggered by those events.

## Key Concepts of Event-Driven Architecture:

Events: Events are notifications of important occurrences in the system. These occurrences can be user actions, changes in state, external signals, or even errors.

Publish-Subscribe: EDA often involves the use of publish-subscribe mechanisms, where events are published by senders and subscribed to by interested receivers.

Decoupling: EDA promotes loose coupling between components. Publishers and subscribers don't need to know about each other's existence. This leads to modular, maintainable, and scalable systems.

Asynchronous Communication: EDA facilitates asynchronous communication, enabling components to process events independently without waiting for immediate responses.

## Advantages of Event-Driven Architecture:

Scalability: EDA allows for scalable systems as components can be added or removed without disrupting the entire system. Events can be processed in parallel.

Modularity: Components are isolated, making it easier to develop, test, and maintain individual parts of the system.

Real-Time Responsiveness: EDA supports real-time updates and reactions to events, providing a more dynamic and responsive system.

Flexibility: New components can be added to react to specific events, promoting extensibility and adaptability.

Event Sourcing: EDA can be used for event sourcing, where the system's state is derived from a sequence of events.

## Use Cases of Event-Driven Architecture:

Notifications and Alerts: Systems can generate notifications or alerts based on specific events, such as sending emails when an order is placed.

Real-Time Analytics: Events can trigger real-time data analysis and reporting.

Microservices Communication: In microservices architectures, events enable communication between services without direct dependencies.

IoT Applications: EDA is often used in IoT systems where devices generate events based on sensor data.

Workflow Automation: Events can drive workflows and process automation, such as approving a purchase order when specific conditions are met.

## Example:

Consider an e-commerce platform. When a user places an order, an "Order Placed" event is generated.

This event can trigger various actions, such as updating inventory, sending a confirmation email, and initiating payment processing. Different services in the system can subscribe to this event and react accordingly, without needing to be tightly coupled.

In summary, Event-Driven Architecture promotes loosely coupled, scalable, and responsive systems by focusing on event propagation and reaction. It's useful for scenarios requiring real-time updates, modular development, and extensibility in distributed systems.

# Event Sourcing

```
+----------------------------------+
|                                  |
|       Command Handlers               |
|                                  |
+----------------------------------+
            |
            | Generates Events
            |
            v
+----------------------------------+
|                                  |
|        Event Store                    |
|                                  |
+----------------------------------+
            |
            | Stores Events
            |
            v
+----------------------------------+
|                                  |
|       Event Handlers                  |
|                                  |
+----------------------------------+
            |
            | Processes Events
            |
            v
```

```
+---------------------------------+
|                          |
|       Aggregate State          |
|                          |
+---------------------------------+
```

In system design, Event Sourcing is a pattern that involves capturing changes in the state of an application as a sequence of events.

Instead of storing the current state of an object or system, Event Sourcing maintains a log of events that have occurred over time.

These events are used to reconstruct the current state at any point in time. This pattern offers several benefits and is particularly useful in scenarios where data history, audit trails, and complex state transitions are important.

## Key Concepts of Event Sourcing:

Events: Events represent significant occurrences or state changes in the system. Each event is a record of what happened, including the type of event and the relevant data.

Event Log: Events are stored in an event log or event store. The event log serves as the source of truth and can be used to reconstruct the current state of the system.

Immutable: Events are immutable once recorded. They cannot be modified, which ensures data integrity and traceability.

State Reconstruction: The current state of the system is reconstructed by replaying the events in the event log. This allows the system to "time-travel" to any point in its history.

## Advantages of Event Sourcing:

Historical Audit: Event Sourcing provides a complete audit trail of all changes and actions taken in the system, making it suitable for compliance and debugging.

Complex State Transitions: Event Sourcing is well-suited for systems with complex state transitions, as events capture the sequence of changes.

Flexible Queries: Data can be restructured and optimized for various queries, enabling better performance for specific use cases.

Evolution of Schema: As events are immutable, changes to the system can be introduced without affecting the existing event records.

Parallel Processing: Events can be processed in parallel, improving performance for applications with high event volumes.

## Use Cases of Event Sourcing:

Financial Systems: Recording financial transactions and maintaining an accurate history for audit purposes.

Supply Chain Management: Tracking the movement of goods and inventory changes over time.

Healthcare Systems: Capturing patient records, diagnoses, treatments, and other medical events.

Gaming: Storing game events, player interactions, and progress for multiplayer and single-player games.

Collaborative Systems: Recording user actions and document changes in collaborative platforms.

## Example:

Consider an e-commerce application using Event Sourcing. Instead of directly updating the stock quantity when an order is placed, the system records an "Order Placed" event with relevant data.

This event is stored in the event log. Subsequent events, such as "Order Shipped" and "Order Cancelled," are also recorded. The current stock quantity can be derived by replaying these events and calculating the resulting stock changes.

In summary, Event Sourcing is a pattern that captures changes in a system's state as a sequence of events. It's useful for maintaining an accurate audit trail, supporting complex state transitions, and enabling historical analysis.

# Command Query Responsibility Segregation (CQRS)

```
+-------------------------------+
|                               |
|        Command Side           |
|                               |
+-------------------------------+
            |
            | Sends Commands
            |
            v
+-------------------------------+
|                               |
|      Command Handlers         |
|                               |
+-------------------------------+
            |
            | Processes Commands
            |
            v
+-------------------------------+
|                               |
|        Event Store            |
|                               |
+-------------------------------+
```

In system design, Command Query Responsibility Segregation (CQRS) is a pattern that separates the responsibilities of handling read and write operations in an application. Instead of using a single model to handle both commands (write operations) and queries (read operations), CQRS advocates the use of separate models for each type of operation. This separation can lead to improved performance, scalability, and flexibility in complex systems.

```
+-------------------------------------+
|                           |
|         Query Side              |
|                           |
+-------------------------------------+
           |
           | Issues Queries
           |
           v
+-------------------------------------+
|                           |
|       Query Handlers             |
|                           |
+-------------------------------------+
           |
           | Processes Queries
           |
           v
+-------------------------------------+
|                           |
|         Read Model              |
|                           |
+-------------------------------------+
```

## Key Concepts of CQRS:

Command Model: The command model handles write operations, such as creating, updating, or deleting data. It focuses on enforcing business rules and maintaining data consistency.

Query Model: The query model is optimized for read operations, providing a denormalized view of the data that is tailored to specific query requirements. It aims to deliver optimal performance for querying data.

Data Duplication: CQRS often involves duplicating data between the command and query models. This allows each model to be designed and optimized independently.

Asynchronous Communication: Commands and events generated by write operations are often processed asynchronously by the command model. Events can trigger updates to the query model.

## Advantages of CQRS:

Performance Optimization: By separating read and write operations, each model can be optimized for its specific purpose. Query performance can be enhanced without impacting the write side.

Scalability: The command and query models can be scaled independently based on their respective workloads.

Flexibility: The query model can be adapted and denormalized to meet the specific needs of various read scenarios, improving responsiveness.

Complexity Management: CQRS can simplify the design of complex systems by allowing different models to evolve independently.

Data Consistency: By focusing on enforcing business rules and consistency in the command model, data integrity can be better maintained.

Support for Event Sourcing: CQRS aligns well with Event Sourcing, as events generated by write operations can be captured and processed by the command model.

## Use Cases of CQRS:

Real-Time Analytics: CQRS can optimize the query model to support real-time analytics and reporting.

E-Commerce: Separating read and write operations can improve the performance of product catalog browsing while maintaining data consistency in the order processing.

Collaborative Systems: In systems with heavy read and write workloads, CQRS can balance the demands on different parts of the application.

### Example:

Consider a banking application using CQRS. When a customer makes a fund transfer (a write operation), the command model validates the transaction, debits the source account, and credits the destination account. The event generated by this transaction is captured and processed asynchronously. The query model, responsible for displaying account balances, is updated with the new balance.

In summary, CQRS is a pattern that advocates the separation of responsibilities between handling write and read operations in an application. It offers advantages in terms of performance optimization, scalability, and flexibility, especially in systems with complex requirements and varying workloads. However, CQRS also introduces the complexity of managing data duplication and eventual consistency between the command and query models. Its adoption should be carefully considered based on the specific needs of the system.

## Saga Pattern

```
+---------------------------------+
|                          |
|      Saga Coordinator       |
|                          |
+---------------------------------+
          |
          | Orchestrates Sagas
          |
              v
     +--------------------------------+
     |                                |
     |         Saga Participants          |
     |                                |
     +--------------------------------+
       /          |          |          \
      /           |          |           \
     /            |          |            \
    |             |          |             |
    |  +---------+  +---------+  +---------+
    |  |  Step 1 |  |  Step 2 |  |  Step 3 |
    |  |         |  |         |  |         |
    |  +---------+  +---------+  +---------+
    |             |          |             |
     \            |          |            /
      \           |          |           /
       \          |          |          /
     +--------------------------------+
     |                                |
     |     External Systems or Events    |
     |                                |
     +--------------------------------+
```

In system design, the Saga pattern is a technique used to manage long-running and distributed transactions in a way that ensures data consistency across multiple services or components. Traditional ACID transactions might not be suitable for distributed systems due to their inherent limitations.

The Saga pattern addresses this challenge by breaking down a complex transaction into a series of smaller, localized transactions known as "saga steps."

Each step represents a single operation and is associated with its own compensating action in case of failures.

## Key Concepts of the Saga Pattern:

Saga Steps: A saga is composed of multiple saga steps, each representing a single transactional operation. Each step has a corresponding compensating action that can undo the effects of the operation.

Atomicity and Consistency: The saga ensures that each individual step is atomic (it either fully completes or fully compensates) and that the overall saga maintains data consistency.

Orchestrator: An orchestrator, often a dedicated service, coordinates the execution of saga steps and manages the progression of the saga. It decides which steps to execute, monitors their progress, and handles compensating actions if needed.

Event-Driven: Communication between saga steps is typically event-driven. Each step generates events to signal its completion or failure, and the orchestrator listens for these events to decide the next steps.

## Advantages of the Saga Pattern:

Distributed Transaction Management: Saga helps manage transactions that span multiple services or components in a distributed system.

Fault Tolerance: Saga handles failures gracefully by allowing for compensating actions to undo partial or failed steps.

Data Consistency: Saga ensures that the overall system remains consistent despite failures during the transaction process.

Decoupling: Saga reduces tight coupling between services by orchestrating interactions through events.

## Use Cases of the Saga Pattern:

E-Commerce: Managing order placement, payment processing, inventory updates, and shipping in an e-commerce system.

Travel Booking: Coordinating flights, hotels, and car rentals when a user makes a travel booking.

Microservices Architecture: Ensuring data consistency across multiple microservices in a complex application.

### Example:

Consider an e-commerce application where a user places an order.

The saga involves several steps, including deducting payment, updating inventory, and sending order confirmation. If any step fails, the saga orchestrator triggers compensating actions, such as refunding payment or restocking inventory. This ensures that even if one step fails, the system remains consistent.

In summary, the Saga pattern is a strategy for managing distributed transactions in a way that ensures data consistency despite failures. It uses a series of localized transactions and compensating actions to maintain the integrity of the overall system. While powerful, implementing sagas requires careful consideration of the system's architecture, event handling, and potential complexities associated with compensating actions.

## Push vs Pull

In system design, "Push" and "Pull" are two contrasting approaches to data or event distribution between components, services, or systems. These approaches determine how information is transmitted and when it's initiated. Each approach has its own advantages and use cases:

Push Approach: In the "Push" approach, the sender actively pushes data or events to one or more recipients without waiting for the recipients to request or poll for the information. When new data or an event becomes available, the sender initiates the communication and delivers it to the recipients.

### Advantages of Push:

```
+---------------------+
|     Data Source     |
|                     |
|      +-------+      |
|      | Data  |      |
|      | Event |      |
|      +-------+      |
|          |          |
|          |          |
|          v          |
|     +-------+       |
|     |  Push |       |
|     |  Data |       |
|     +-------+       |
+---------------------+
```

Real-time Updates: Push enables real-time or near-real-time updates, making it suitable for scenarios that require immediate communication, such as live dashboards or instant notifications.

Event-Driven Architecture: Push supports event-driven designs where components react to events as they occur.

Lower Latency: Push can reduce communication latency since the information is delivered as soon as it's available.

## Use Cases of Push:

Instant Messaging Applications: Messages are pushed to users in real-time.

Notifications: Sending alerts, reminders, or updates to users.

Real-Time Collaborative Editing: Updates are pushed to collaborators as others make changes.

Streaming Platforms: Delivering continuous data streams to consumers.

Pull Approach: In the "Pull" approach, the recipients actively poll or request data or events from the sender or a central source. Recipients periodically check for updates, and when they find new information, they retrieve it.

## Advantages of Pull:

```
+----------------------+
|     Data Source      |
|                    |
|     +-------+       |
|     | Data  |       |
|     |  Store|       |
|     +-------+       |
|        |          |
|        |          |
|        v          |
|    +-------+       |
|    |  Pull |       |
|    |  Data |       |
|    +-------+       |
+----------------------+
```

Controlled Consumption: Recipients can control when and how often they retrieve data, preventing information overload.

Easier Load Management: Since recipients initiate requests, load spikes on the sender's side can be managed better.

Lower Bandwidth Usage: Data is only fetched when needed, reducing unnecessary network traffic.

## Use Cases of Pull:

Batch Data Processing: Collecting and processing data in batches at intervals.

Resource Monitoring: Pulling data from remote sensors or monitoring systems.

Polling APIs: Fetching updates from external APIs at scheduled intervals.

Choosing Between Push and Pull: The choice between push and pull depends on the specific requirements of your system and use case:

- Push is suitable for real-time or event-driven scenarios where immediate updates are essential.
- Pull is useful when recipients need to manage their own consumption rate, or when data updates are less frequent.

In many cases, a combination of push and pull approaches can be used to achieve the desired behavior. For example, a system might use push for real-time updates and pull for less time-sensitive data retrieval.

# CACHING

## Caching Layers

In system design, various caching mechanisms are employed at different layers of a software architecture to improve performance, reduce latency, and optimize resource utilization.

Let's explore the differences between client caching, CDN caching, web server caching, database caching, and application caching, along with their respective benefits:

### Client Caching:

Client caching occurs on the user's device (browser, mobile app) and involves storing copies of frequently accessed resources locally. These resources include HTML, CSS, JavaScript, images, and other media files.

**Useful For:** Reducing page load times for returning users, minimizing server load, and improving user experience.

**Benefits:** Faster load times for returning users, reduced bandwidth usage, and decreased server load.

### CDN (Content Delivery Network) Caching:

CDN caching involves replicating and storing content across a network of geographically distributed servers. These servers serve cached content to users based on their proximity, reducing latency and improving content delivery.

**Useful For:** Accelerating content delivery to users across different regions, improving global performance, and handling traffic spikes.

**Benefits:** Faster content delivery, reduced load on origin servers, improved availability, and better scalability.

## Web Server Caching:

Web server caching involves storing static and dynamic content on the web server itself. It can be configured to serve cached content to users based on specific conditions, such as expiration time or request headers.

**Useful For:** Enhancing performance by serving frequently accessed content directly from the server, reducing the load on application and database servers.

**Benefits:** Faster response times, reduced load on backend systems, improved scalability, and better handling of traffic spikes.

## Database Caching:

Database caching involves storing frequently accessed database query results in memory to reduce the need for repeated database queries. It's particularly beneficial for read-heavy workloads.

**Useful For:** Optimizing database performance and reducing query response times for read operations.

**Benefits:** Faster query execution, reduced database load, improved scalability, and enhanced application responsiveness.

## Application Caching:

Application caching involves caching computed results or objects within the application's memory. It's often used for storing data that's expensive to compute or retrieve from external sources.

**Useful For:** Improving application performance by avoiding redundant computations or external API calls.

**Benefits:** Faster response times, reduced resource consumption, and improved overall application efficiency.

## Key Takeaways:

- **Client Caching** improves user experience by storing resources on the user's device.

- **CDN Caching** enhances content delivery by replicating content across a network of servers.
- **Web Server Caching** optimizes response times by serving cached content directly from the web server.
- **Database Caching** speeds up query execution by storing frequently accessed data in memory.
- **Application Caching** improves application performance by caching computed results.

In summary, caching mechanisms play a critical role in system design by enhancing performance, reducing latency, and optimizing resource utilization at various layers of an architecture. Each type of caching is designed to address specific performance bottlenecks and contribute to a more efficient and responsive system.

# Caching Strategies

## Refresh Ahead (Read-Ahead) Caching:

Refresh ahead caching involves proactively fetching data from the backend or storage before it's requested by users. This strategy aims to reduce the latency associated with retrieving data on demand.

**Useful For:** Improving read performance by anticipating user needs and minimizing the delay caused by fetching data.

**Benefits:** Reduced read latency, improved user experience, and efficient utilization of cache.

## Write Behind Caching:

Write-behind caching defers write operations to the cache and asynchronously updates the backend or storage later. This strategy is particularly useful when write operations need to be optimized for speed and responsiveness.

**Useful For:** Enhancing write performance by reducing the immediate impact of write operations on the backend.

**Benefits:** Improved write throughput, reduced latency for write-intensive workloads, and efficient utilization of resources.

## Write-Through Caching:

Write-through caching involves immediately writing data to both the cache and the backend storage when a write operation occurs. This ensures data consistency between the cache and the backend.

**Useful For:** Maintaining data consistency and integrity between the cache and the backend storage.

**Benefits:** Data consistency, reduced risk of stale data, and read-after-write consistency.

## Cache-Aside (Lazy Loading) Caching:

Cache-aside caching involves allowing the application to interact directly with the cache. When data is needed, the application checks the cache first. If the data is not found in the cache, it's fetched from the backend and then stored in the cache for future use.

**Useful For:** Providing fine-grained control over cached data and minimizing the risk of stale data.

**Benefits:** Control over cache management, reduced risk of stale data, and efficient utilization of cache.

## Key Takeaways:

- **Refresh Ahead Caching** proactively fetches data before it's requested to reduce read latency.
- **Write Behind Caching** defers write operations to the cache and asynchronously updates the backend.
- **Write-Through Caching** maintains data consistency by immediately updating both the cache and the backend.
- **Cache-Aside Caching** allows the application to manage cache interactions directly.

In summary, these caching strategies offer different ways to optimize data access and improve performance. The choice of strategy depends on the specific requirements of the application, the trade-offs between data consistency and performance, and the data access patterns.

# Cache Cleanup

In system design, both cache invalidation and cache eviction are strategies used to manage and optimize the behavior of caches. They serve different purposes and are used in different situations:

## Cache Invalidation:

**Purpose**: Cache invalidation is a strategy used to remove specific items or entries from the cache when the data they represent becomes outdated or no longer valid.

**Usage**: Cache invalidation is typically used when you have a way to detect changes in the underlying data source. When the data in the source changes, you invalidate the corresponding cached item(s) to ensure that future requests will fetch fresh, up-to-date data.

**Use Cases**: It's especially useful in scenarios where data changes frequently and you want to maintain cache consistency. For example, in a web application, you might invalidate the cache for a user's profile when they update their information.

**Advantages**:

- Ensures that cached data remains up to date.
- Reduces the risk of serving stale or outdated data to users.
- Can be more efficient than constantly refreshing the entire cache.

## Cache Eviction:

**Purpose**: Cache eviction is a strategy used to remove items or entries from the cache to make room for new data when the cache becomes full.

**Usage**: Cache eviction is employed when you have a limited amount of memory or storage allocated for caching. When the cache reaches its capacity, you need to decide which items to remove to accommodate new data.

**Use Cases**: It's particularly useful in scenarios where you need to balance between maximizing cache hits (serving data from the cache) and minimizing cache misses (removing the least valuable items when the cache is full).

**Advantages**:

- Helps manage memory or storage resources efficiently.
- Ensures that the cache doesn't grow indefinitely, preventing potential performance problems.

## Key Differences:

- Purpose: Cache invalidation focuses on keeping the cache's content up to date and consistent with the underlying data, while cache eviction focuses on managing the cache's size and preventing it from becoming overloaded.
- Trigger: Cache invalidation is typically triggered by changes in the data source, while cache eviction is triggered by the cache reaching its capacity or a predefined eviction policy.
- Data Removal: Cache invalidation removes specific items from the cache based on events or criteria, whereas cache eviction removes items based on a defined policy, such as LRU (Least Recently Used) or LFU (Least Frequently Used).

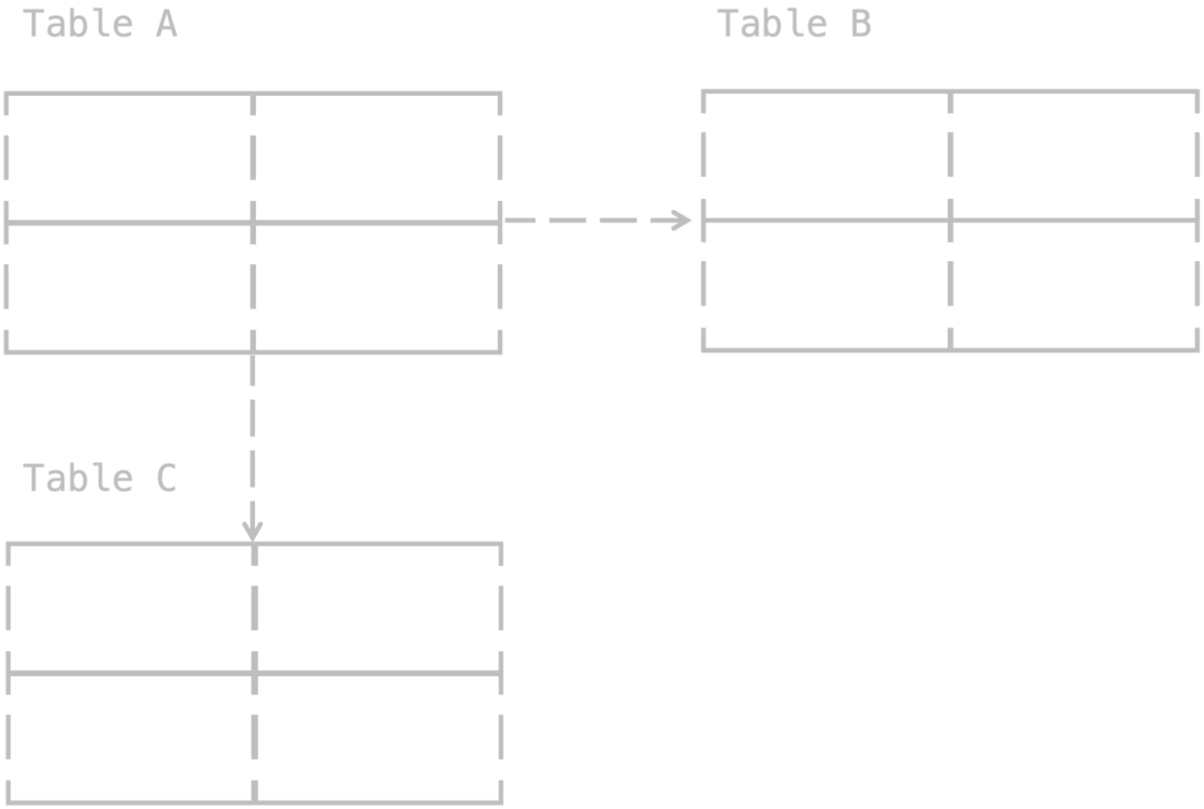
In practice, both strategies are often used in combination to create an efficient and well-behaved caching system. Cache invalidation ensures data consistency, while cache eviction manages the cache's resource consumption. The choice of which strategy or combination to use depends on the specific requirements and characteristics of your system.

# DATABASES

## Normalization vs Denormalization

Normalization and denormalization are two opposing database design techniques used to optimize database schemas for different purposes. They have distinct advantages and trade-offs in terms of data integrity, storage efficiency, and query performance:

### Normalization:



Purpose:

**Minimize Redundancy**: The primary goal of normalization is to minimize data redundancy by breaking down a database into smaller, related tables, each containing a specific set of attributes. This reduces the risk of data anomalies and ensures data integrity.

Use Cases:

Normalization is commonly used in transactional databases where data integrity is critical. It's suitable for systems that require strict adherence to the principles of the relational model.

Advantages:

**Data Integrity**: Normalization reduces the chances of data anomalies like update anomalies, insert anomalies, and delete anomalies.

**Space Efficiency**: It minimizes storage space by storing data in a structured and compact manner.

**Easier Maintenance**: Changes to data are easier to manage since data is stored in a structured and modular way.

Drawbacks:

**Query Performance**: Normalized databases can suffer from complex joins when retrieving data, which may impact query performance. JOIN operations can be resource-intensive.

## Denormalization:

```
 Denormalized Table

┌──────────┬──────────┬──────────┬──────────┬──────────┬──────────┐
│          │          │          │          │          │          │
├──────────┼──────────┼──────────┼──────────┼──────────┼──────────┤
│          │          │          │          │          │          │
└──────────┴──────────┴──────────┴──────────┴──────────┴──────────┘
```

Purpose:

**Improve Query Performance**: Denormalization involves combining tables and introducing redundancy to optimize query performance. It's done to speed up read-heavy operations by reducing the need for complex joins.

Use Cases:

Denormalization is often used in data warehousing, reporting, and analytics databases where read operations are frequent, and query performance is a top priority.

Advantages:

**Query Performance**: Denormalized databases typically perform better for read-heavy workloads because they minimize JOIN operations.

**Simplified Queries**: Data retrieval is more straightforward as data is often stored in a way that matches the application's query patterns.

Drawbacks:

**Data Integrity**: Denormalization increases the risk of data anomalies since redundant data can lead to inconsistencies if not properly maintained.

**Storage Overhead**: Redundant data occupies more storage space, which can be a concern for large datasets.

**Update Anomalies**: Updates to denormalized data require careful management to ensure consistency.

Use Cases:

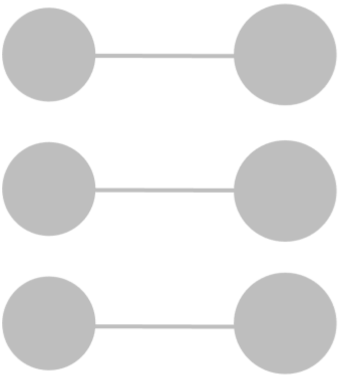
- Use normalization when data integrity is paramount, such as in transactional databases, financial systems, or systems with strict compliance requirements.
- Use denormalization when query performance is a top priority, and you can accept a controlled level of data redundancy and are willing to implement mechanisms to maintain data consistency.

In practice, many database systems strike a balance between normalization and denormalization to achieve both data integrity and query performance. Techniques like indexing, caching, and materialized views are also used to optimize query performance in normalized databases without resorting to full denormalization. The choice between these approaches depends on the specific requirements of the application and the database design.

## NoSQL Databases

In the context of database systems, different types of databases are designed to handle various data models and use cases. Here's a comparison of key-value stores, document stores, wide-column stores, and graph databases, along with their respective use cases and advantages:

## Key-Value Store:



Data Model:

Stores data as a collection of key-value pairs, where each key is unique and corresponds to a value.

Use Cases:

**Caching**: Ideal for caching frequently accessed data due to fast read and write operations.

**Session Management**: Useful for storing user session information.

**Distributed Systems**: Often used in distributed and scalable systems due to their simplicity.

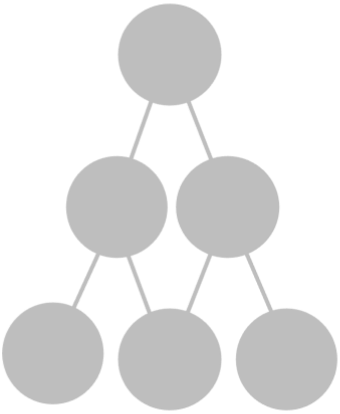
Advantages:

**High Performance**: Extremely fast read and write operations, especially when data access is predictable.

**Scalability**: Easily scalable horizontally to handle increased load.

**Flexibility**: Can store arbitrary data structures as values.

## Document Store:



Data Model**:**

Stores data as semi-structured documents (e.g., JSON, XML) with unique identifiers.

Use Cases**:**

**Content Management**: Suitable for content management systems and content-heavy applications.

**Catalogs**: Used in e-commerce applications for product catalogs.

**Real-time Analytics**: Supports flexible schema design for analytics.

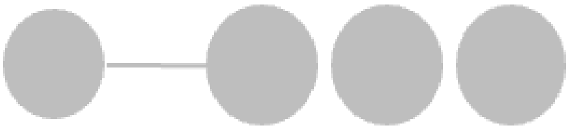
Advantages**:**

**Schema Flexibility**: Allows for dynamic and evolving data schemas.

**Rich Querying**: Supports querying on document attributes.

**Horizontal Scalability**: Can handle large volumes of data and traffic.

## Wide-Column Store (Column-Family Store):



Data Model:

Organizes data into columns rather than rows, with a focus on optimizing read-heavy workloads.

Use Cases:

**Time-Series Data**: Suitable for storing time-series data like logs and sensor data.

**Analytics**: Used in data warehousing and analytical applications.

**Content Management**: Can be used for content storage when querying involves selecting a subset of columns.

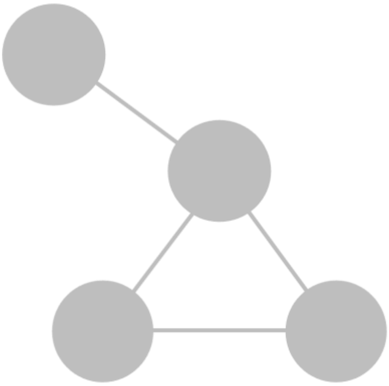
Advantages:

**High Read Performance**: Optimized for read-heavy workloads due to columnar storage.

**Scalability**: Scales well for analytical use cases with large datasets.

**Compression**: Efficiently compresses data, reducing storage requirements.

## Graph Database:



Data Model:

Represents data as nodes, edges, and properties, allowing for efficient traversal of relationships.

Use Cases:

**Social Networks**: Ideal for modeling social network connections.

**Recommendation Engines**: Used for personalized recommendations based on user behavior.

**Knowledge Graphs**: Suitable for representing and querying complex relationships in data.

Advantages:

**Relationship Handling**: Efficiently manages and traverses relationships between data entities.

**Complex Queries**: Supports complex graph-based queries.

**Contextual Insights**: Provides insights into relationships and connections within data.

### Use Cases:

- Choose a **Key-Value Store** for simple data retrieval and caching scenarios where read and write performance are critical, and the data structure is relatively straightforward.
- Opt for a **Document Store** when dealing with semi-structured data, dynamic schemas, and applications where flexibility in data representation is required.
- Consider a **Wide-Column Store** when handling large volumes of data, especially for analytics or time-series data, where read performance and horizontal scalability are essential.
- Select a **Graph Database** when your data involves complex relationships, and you need efficient traversal of these relationships to answer queries or make recommendations.

## ACID vs BASE

ACID and BASE are two contrasting sets of properties that define different approaches to data consistency and system behavior in distributed database systems:

### ACID (Atomicity, Consistency, Isolation, Durability):

**Purpose**:

ACID properties are designed to ensure strong data consistency and reliability in database transactions.

**Atomicity**:

Ensures that a transaction is treated as a single, indivisible unit of work, and it is either fully completed or fully rolled back in case of failure.

**Consistency**:

Guarantees that a transaction brings the database from one consistent state to another, preserving integrity constraints.

**Isolation**:

Ensures that concurrent transactions do not interfere with each other and that they appear to execute serially, even when running concurrently.

**Durability**:

Ensures that once a transaction is committed, its changes are permanent and will survive system failures.

## BASE (Basically Available, Soft state, Eventually consistent):

**Purpose**:

BASE properties prioritize availability and partition tolerance over strict consistency, making it more suitable for distributed systems with a focus on high availability and fault tolerance.

**Basically Available**:

The system remains available for read and write operations, even in the presence of network partitions or failures.

**Soft state**:

The system's state may be temporarily inconsistent due to eventual consistency models, where updates may take time to propagate across the system.

**Eventually Consistent**:

Over time, the system will converge to a consistent state as all changes propagate and conflicts are resolved.

## Use Cases:

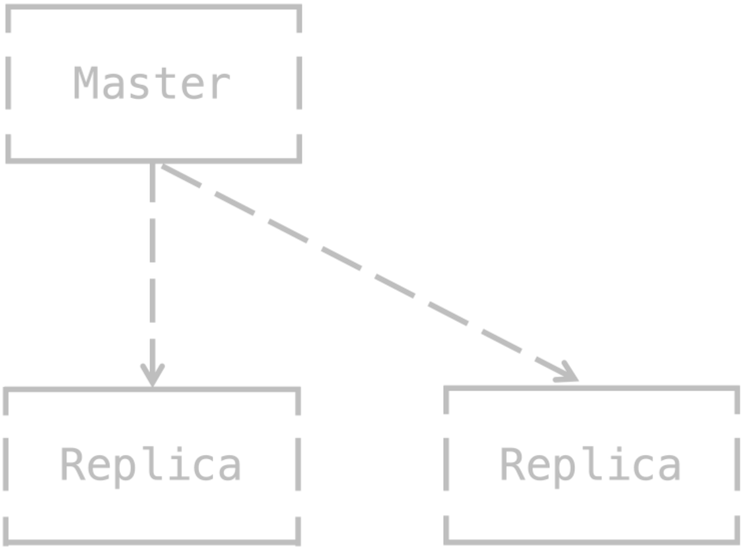
- **ACID** is suitable for applications where data integrity and reliability are of utmost importance, such as financial systems, healthcare databases, and scenarios where correctness and consistency are non-negotiable.
- **BASE** is useful in distributed systems where high availability and fault tolerance are critical, and some degree of temporary inconsistency can be tolerated. It is commonly used in NoSQL databases, content delivery networks, and systems that prioritize scalability and responsiveness over strict consistency.

## Key Differences:

- ACID prioritizes strong consistency and data integrity, while BASE prioritizes availability and fault tolerance.
- ACID guarantees immediate consistency and durability, whereas BASE allows for temporary inconsistencies that will eventually be resolved.
- ACID is often used in traditional relational databases, while BASE is commonly associated with NoSQL and distributed databases.

The choice between ACID and BASE depends on the specific requirements of the system and the trade-offs that can be made. In practice, some systems may use a combination of both approaches, applying ACID principles where strong consistency is necessary and BASE principles where high availability and scalability are required.

## Replication



Replication in database systems involves creating and maintaining copies (replicas) of data on multiple servers. It serves several key purposes:

**High Availability**:

> Ensures system uptime by providing redundancy. If one server fails, another replica can take over.

**Load Balancing**:

> Distributes read traffic among replicas, improving system performance by reducing the load on the primary database.

**Disaster Recovery**:

> Acts as a backup in case of data corruption, loss, or catastrophic failures.

**Geographic Distribution**:

> Reduces latency for users in different regions by placing data closer to them.

**Read Scaling**:

> Offloads read-heavy workloads to replicas, freeing the primary database for write operations.

**Offline Processing**:

Supports data analysis, reporting, and other offline tasks without impacting the primary database.

Replication enhances data availability, reliability, and performance in database systems, making it a fundamental component of many modern architectures.

## Sharding and partitioning

Sharding and partitioning are database design strategies used to manage and distribute large volumes of data, but they have distinct characteristics and purposes:

### Sharding:

```
+-------------------------------+

|   Sharded Database Instance    |

|                               |

|   +---------+   +---------+    |

|   | Shard 1 |   | Shard 2 |    |

|   |         |   |         |    |

|   +---------+   +---------+    |

|                               |

+-------------------------------+



+-------------------------------+

|   Sharded Database Instance    |

|                               |

|   +---------+   +---------+    |

|   | Shard 3 |   | Shard 4 |    |
|   |      | |      |   |
|  +---------+  +---------+   |
|                        |
+----------------------------------+
```

Purpose: Sharding is a technique used to horizontally partition a single, large database into smaller, more manageable pieces called shards. Each shard holds a distinct subset of the data and operates as an independent database.

Use Cases:

- **Scalability**: Sharding is primarily used to scale a database horizontally to handle increased data volume, read and write traffic, and storage requirements.
- **Data Isolation**: Sharding can isolate different types of data, customer data, or data from different geographical regions.

Advantages:

- **Scalability**: Sharding allows for linear scalability by distributing data and queries across multiple servers or clusters.
- **Performance**: Improved query performance by reducing the amount of data each shard needs to manage.
- **Isolation**: Data isolation can improve security and compliance by keeping sensitive data on separate shards.

## Partitioning:

```
+-------------------------------------+
|        Partitioned Table          |
|                           |
|  +-----------+  +-----------+   |
|    | Partition 1 | Partition 2 |    |
|    |           |           |    |
|    +-----------+    +-----------+    |
|                                |
+---------------------------------+
```

Purpose: Partitioning is a technique used to divide a database or table into smaller, manageable segments called partitions based on certain criteria, such as ranges of values or hash functions. Partitions still belong to the same database but are organized differently for improved data management and query performance.

Use Cases:

**Data Organization**: Partitioning helps organize data within a single database for better management, maintenance, and query optimization.

**Data Distribution**: It can also be used to distribute data across storage devices or file systems for efficiency.

Advantages:

**Query Performance**: Partitions can improve query performance by allowing the database to skip scanning irrelevant partitions when executing queries.

**Data Management**: Easier management of large tables, backups, and archiving by segmenting data into smaller partitions.

**Maintenance**: Partitioning facilitates maintenance tasks, such as archiving old data or optimizing storage.

## Key Differences:

**Scope**: Sharding typically involves distributing data across multiple independent databases or database clusters, while partitioning operates within a single database.

**Independence**: Shards in a sharded system often function as independent databases, while partitions in a partitioned table are still part of the same database and share the same schema.

**Use Cases**: Sharding is primarily used for achieving horizontal scalability and data isolation in distributed systems, while partitioning is used for organizing data within a single database for improved data management and query optimization.

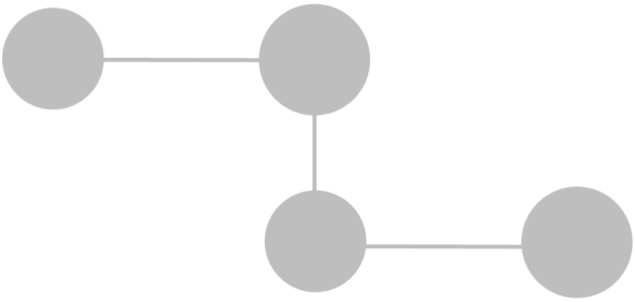
## Use Cases:

Use **sharding** when you need to horizontally scale your database across multiple database instances to handle large data volumes and high traffic loads in a distributed environment.

Use **partitioning** when you want to organize and optimize data within a single database, making it easier to manage, improve query performance, and facilitate maintenance tasks.

Both sharding and partitioning are valuable techniques for managing data in databases, and the choice between them depends on your specific scalability and data management requirements. In some cases, you may even use both techniques in combination to achieve the desired results.

# Data Modeling Factors



When starting to create a data model for an application or when designing one in an interview, consider the following important factors:

**Normalization**:

> Decide on the level of data normalization, striking a balance between reducing redundancy and optimizing query performance.

**Data Integrity**:

> Enforce data integrity constraints, such as primary keys, foreign keys, unique constraints, and check constraints, to maintain data accuracy and consistency.

**Indexes**:

> Identify columns that require indexing to improve query performance, and consider the type of indexes (e.g., B-tree, hash, full-text) that best suit each use case.

**Partitioning**:

> If dealing with large datasets, consider partitioning strategies to manage and query data efficiently.

**Denormalization**:

> Determine if limited denormalization is necessary to optimize read-heavy operations, but be cautious about potential data inconsistencies.

**Concurrency Control**:

Consider how the database will handle concurrent access and implement appropriate locking or isolation mechanisms.

**Scalability**:

Plan for future scalability by designing a schema that can accommodate increased data volume and user load.

**Joins**:

Be mindful of the impact of indexes and join operations on query performance, and strike a balance between them.

**Normalization vs. Performance**:

Be prepared to justify your design decisions, especially when balancing normalization for data integrity and denormalization for performance.

**Optimizing for Write Operations**:

Consider how the data model will handle write-intensive operations and implement strategies for minimizing contention.

Remember that designing an effective data model requires a balance between various factors, including data integrity, query performance, and scalability, and it often involves trade-offs. Clear communication and justifications for your design choices are crucial in both real-world projects and interview scenarios.

# Storage

Block storage, file storage, object storage, and redundant disk arrays are different storage technologies, each with its own characteristics and use cases:

## Block Storage:

```
+------------------+
|   Block 1        |
|   (Data)         |
+------------------+
|   Block 2        |
|   (Data)         |
+------------------+
|   Block 3        |
|   (Data)         |
+------------------+
|   Block 4        |
|   (Data)         |
+------------------+
```

Data Structure:

Divides data into fixed-sized blocks, typically with a unique identifier.

Use Cases:

Used in storage area networks (SANs) and cloud computing to provide high-performance, low-latency storage for applications and databases.

Ideal for scenarios where precise control over data placement and direct access to individual blocks is required.

Advantages:

Low-Level Access: Allows direct access to storage blocks, making it suitable for databases and virtualization.

Performance: Offers high I/O performance and low latency.

## File Storage:

```
+-----------------+
|  Directory 1   |
|  +-------------+
|  |  File 1    |
|  |  File 2    |
|  +-------------+
+-----------------+
|  Directory 2   |
|  +-------------+
|  |  File 3    |
|  | Directory 3|
|  |  +----------+
|  |  |  File 4 |
|  |  +----------+
|  +-------------+
+-----------------+
```

Data Structure:

Organizes data into files and directories, providing a hierarchical structure.

Use Cases:

Commonly used in traditional file servers, network-attached storage (NAS), and distributed file systems.

Suitable for shared file access and data organization where files need to be accessed concurrently by multiple users or applications.

Advantages:

File-Level Access: Provides a familiar file system interface for organizing and accessing data.

Sharing: Supports concurrent access by multiple users or applications.

## Object Storage:

```
+--+
|  |
|  |
+--+

+-----+
|     |
|     |
+-----+

  /\
 /  \
/____\
```

Data Structure:

Stores data as objects, each with its metadata and a unique identifier.

Use Cases:

Frequently used in cloud storage, content delivery, and web applications for storing and serving unstructured data like images, videos, and documents.

Ideal for scalable, globally distributed data storage.

Advantages:

Scalability: Scales horizontally to accommodate vast amounts of unstructured data.

Metadata: Provides rich metadata, making it suitable for organizing and searching large datasets.

## Redundant Disk Arrays (RAID):

Data Structure:

Combines multiple physical hard drives into an array to provide data redundancy and/or improved performance.

Use Cases:

Commonly used in servers and storage systems to enhance data availability and reliability.

Various RAID levels offer different combinations of redundancy and performance.

Advantages:

Data Redundancy: Protects against data loss by duplicating data across multiple drives.

Performance: Some RAID levels can improve read and write performance.

## Why They Are Useful:

**Block Storage** is useful for applications that require low-level, direct access to storage blocks, such as databases and virtualization. It offers high performance and control.

**File Storage** is beneficial when multiple users or applications need to share and organize data using a hierarchical file system structure, as seen in traditional file servers and NAS devices.

**Object Storage** is valuable for storing and serving large volumes of unstructured data on a global scale, making it suitable for cloud storage and content delivery.

**Redundant Disk Arrays (RAID)** are essential for enhancing data availability and reliability in storage systems. Different RAID levels provide various trade-offs between redundancy and performance.

## Idempotent operations

```
+-----------------------+
|                 |
|       Operation       |
|                 |
+-----------------------+
       |
       | Performs
       |
+-----------------------+
|                 |
|       System State     |
|                 |
+-----------------------+
```

In a system design context, an idempotent operation is an operation that can be applied multiple times without changing the result beyond the initial application. In other words, performing the same idempotent operation multiple times has the same effect as performing it once. This property has several important use cases and advantages:

**Fault Tolerance and Reliability**:

> Idempotent operations are often used in distributed systems and network communications. In scenarios where network failures or timeouts can occur, retrying an idempotent operation ensures that the operation is eventually completed successfully without causing unintended side effects.

**Idempotent APIs**:

Idempotence is a desirable property for public APIs and web services. It allows clients to safely repeat requests without fear of unintended consequences. For example, HTTP methods like GET and PUT are idempotent.

**Retry Mechanisms**:

Idempotence simplifies the implementation of retry mechanisms in systems. When an operation fails or times out, it can be retried without worrying about the state or result of previous attempts.

**Consistency in State Transitions**:

In state machines and workflow systems, idempotent transitions ensure that a state change from one state to another can be repeated without affecting the final state. This is essential for maintaining system consistency.

**Memoization**:

Idempotent operations are suitable for caching and memoization. When a result is calculated once, it can be cached and reused for subsequent requests with the same inputs.

**Financial Transactions**:

In financial systems, idempotent operations are used to guarantee that a transaction is processed exactly once, even if there are system failures or retries.

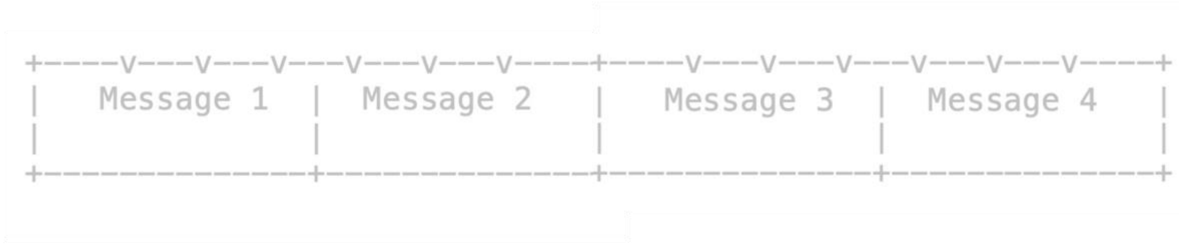
**Exactly once guarantees**

Idempotent semantics are critical in scenarios where operations must be executed exactly once, such as in distributed databases, message queues, and payment processing systems..

In summary, the concept of idempotent operations is essential for building reliable and fault-tolerant systems. It simplifies error handling, enables retries, and ensures that the system remains in a consistent and predictable state, even in the face of network failures or other unforeseen issues.

Idempotence is a valuable property for designing systems that can handle unexpected events gracefully.

# Message Queues



```
+----v---v---v---v---v---v----+----v---v---v---v---v---v----+
|   Message 1   |  Message 2   |   Message 3   |  Message 4   |
|               |              |               |              |
+---------------+--------------+---------------+--------------+
```

In system design, a Message Queue is a communication mechanism that allows different components of a distributed system to exchange messages asynchronously.

It acts as an intermediary buffer that temporarily stores messages before they are processed by the intended recipients. Message Queues play a crucial role in achieving loose coupling, scalability, and resilience in distributed architectures.

## Key Concepts and Benefits of Message Queues:

**Asynchronous Communication:**

Message Queues enable decoupled communication between components. Senders and receivers do not need to be active simultaneously, allowing for asynchronous processing.

**Loose Coupling:**

Components can communicate without direct knowledge of each other's existence or state. This promotes modularity and easier maintenance.

**Scalability:**

Message Queues facilitate load balancing by distributing incoming requests across multiple instances of a service, helping prevent bottlenecks.

**Resilience:**

Messages are stored in the queue until they are processed. This enables systems to handle temporary spikes in traffic or service failures without losing data.

**Event-Driven Architecture:**

Message Queues support event-driven design, where components react to events triggered by incoming messages.

**Cross-System Integration:**

Message Queues facilitate communication between different services, applications, and even across different systems.

**Message Transformation**:

Some Message Queues offer capabilities to transform or enrich messages as they are processed.

## Use Cases:

**Microservices Communication:**

In microservices architectures, Message Queues enable seamless communication between different services, allowing them to work independently.

**Event Sourcing:**

Message Queues can be used to implement event sourcing, capturing and replaying events to recreate system state.

**Batch Processing:**

Message Queues are suitable for handling batch processing jobs, where data processing can be deferred and optimized.

**Task Queues:**

Message Queues can manage task queues, distributing tasks to workers and ensuring efficient utilization of resources.

**Real-time Data Processing:**

In data streaming scenarios, Message Queues can be part of real-time data processing pipelines.

## Example:

Consider an e-commerce application. When a customer places an order, the order service can publish a message to a Message Queue. The payment service subscribes to the queue and processes payment for the order. This decouples the order and payment processes, allowing them to scale independently and ensuring that orders are not lost even if the payment service experiences downtime.

In summary, Message Queues provide an efficient and reliable way for distributed components to communicate asynchronously, promoting loose coupling, scalability, and resilience in various system design scenarios.

## Reactive programming

```
+----------------------+
|                      |
|  Reactive Programming |
|                      |
+----------------------+
       |  |  |
       |  |  |
+-------v---v---v-------+
|  Data Streams          |
|  (Observable, Stream)  |
+------------------------+
       |  |  |
       |  |  |
+-------v---v---v-------+
|  Operators             |
|  (Transformations)     |
+------------------------+
```

Reactive programming is a programming paradigm and design approach that emphasizes the handling of asynchronous data flows and the propagation of changes through data streams.

It's particularly useful for building systems that need to react to and process real-time events, user interactions, or data from various sources. Here's a closer look at the concept and its utility:

## Key Characteristics of Reactive Programming:

**Asynchronous Data Streams**:

Reactive programming revolves around the concept of data streams or sequences of events over time. These streams can represent user input, sensor data, network responses, or any kind of asynchronous data.

**Observer Pattern**:

Reactive programming often employs the Observer pattern, where components (observers) subscribe to data streams (observables) and react to changes as they occur. Observables emit data, and observers respond to it.

**Transformation and Composition**:

Reactive programming provides a rich set of operators to transform, filter, combine, and manipulate data streams. This allows for the creation of complex data processing pipelines with ease.

**Backpressure Handling**:

In scenarios where data streams can produce data faster than they can be consumed, reactive programming frameworks often provide mechanisms for handling backpressure, ensuring that data is processed without overwhelming the system.

## Why Reactive Programming Is Useful:

**Real-Time and Event-Driven Applications**:

Reactive programming is well-suited for building real-time applications and systems that respond to user interactions, sensor data, IoT devices, and other events as they occur.

**Concurrent and Parallel Processing**:

It simplifies the handling of concurrent operations and parallel processing. Multiple asynchronous tasks can be managed in a coordinated and reactive manner.

**Responsive User Interfaces**:

In web and mobile app development, reactive frameworks like React, Angular, and Vue.js allow for the creation of responsive and interactive user interfaces that update in real-time based on user actions and data changes.

**Scalability**:

Reactive systems can be designed to scale horizontally to handle high loads and increased data volumes. They are often used in distributed and microservices architectures.

**Data Transformation and Processing**:

It simplifies complex data transformation and processing tasks, such as filtering, mapping, aggregation, and joining, by providing a rich set of operators.

In summary, reactive programming is a powerful paradigm for designing systems that need to handle asynchronous data, real-time events, and concurrent processing efficiently. It promotes responsiveness, scalability, and maintainability, making it a valuable approach in modern software development.

## Back Pressure Handling

```
+------------------------+
|                    |
|   Data Producer       |
|                    |
+------------------------+
        |  |  |
        |  |  |
+---------v---v---v-------+
|   Data Channel        |
|  (e.g., Queue, Stream) |
+------------------------+
        |  |  |
        |  |  |
+---------v---v---v-------+
|  Data Processor       |
|   (e.g., Consumer)      |
+------------------------+
```

In the context of asynchronous systems and event-driven architectures, the concept of **back pressure** is a critical mechanism for managing the flow of data or events when the rate of production exceeds the rate of consumption or processing.

It's particularly important in scenarios where asynchronous data sources, such as message queues, sensors, or external services, can produce data at varying rates. Here's a closer look at back pressure, its implementation, and its significance:

## Back Pressure Concept:

**Definition**:

> Back pressure is a way for a consumer or downstream component to signal to a producer or upstream component that it cannot process data or events as quickly as they are being produced. It is a form of flow control.

**Purpose**:

> Back pressure ensures that the system remains within its capacity limits, preventing data loss, resource exhaustion, and system instability. It allows the system to adapt to fluctuations in workloads.

## Implementation of Back Pressure:

**Buffering**:

> Maintain a buffer or queue to temporarily store incoming data or events. Control the buffer size or capacity. When the buffer is full, it triggers back pressure, causing the producer to slow down or stop producing until space is available in the buffer.

**Rate Limiting**:

> Implement rate-limiting mechanisms to restrict the rate at which data is consumed or processed. This can be done by specifying a maximum processing rate or limiting the number of requests or events per unit of time.

**Flow Control Protocols**:

Use flow control protocols or mechanisms that are inherent to specific communication protocols or messaging systems. For example, message queuing systems like RabbitMQ and Apache Kafka provide built-in support for back pressure through acknowledgments and consumer offsets.

**Dynamic Scaling**:

Automatically scale the number of processing nodes or resources in response to changing workloads. Scaling up or down helps accommodate increased or decreased data flows without applying back pressure.

# COMMUNICATION

## HTTP & REST

```
  Client        Server

    |             |

    |  Request      |

    |-------------->|

    |             |

    |  Response      |

    |<--------------|

    |             |
```

HTTP (Hypertext Transfer Protocol) **and** REST (Representational State Transfer) are fundamental concepts in web-based system design. Let's explore what they are, how they are implemented, and their utility:

### HTTP (Hypertext Transfer Protocol):

**Definition:**

> HTTP is an application layer protocol used for transmitting and receiving data over the World Wide Web. It serves as the foundation of communication between clients (such as web browsers) and web servers.

**Implementation**:

Request-Response:

HTTP follows a request-response model. A client sends an HTTP request to a server, and the server responds with an HTTP response. Each request and response includes headers and, optionally, a message body.

Methods:

HTTP defines various request methods, including GET (retrieve data), POST (create data), PUT (update data), DELETE (remove data), and more. These methods determine the action to be performed on a resource.

Status Codes:

HTTP responses include status codes, such as 200 (OK), 404 (Not Found), and 500 (Internal Server Error), to indicate the outcome of a request.

Stateless:

HTTP is stateless, meaning each request from a client to a server is independent, and the server does not retain information about previous requests.

**Use Cases**:

Web Browsing:

HTTP is the protocol used for web browsing, allowing users to access and interact with websites.

APIs:

Many web services and APIs use HTTP as the underlying protocol to expose their functionalities, making it possible for clients to access and manipulate data remotely.

Resource Retrieval:

HTTP is used for retrieving resources like web pages, images, videos, and documents.

## REST (Representational State Transfer):

**Definition**:

REST is an architectural style for designing networked applications. It emphasizes a set of constraints for building scalable and stateless web services. RESTful services are designed around resources and follow a client-server model.

**Implementation**:

Resources:

In REST, resources are represented by URIs (Uniform Resource Identifiers), such as URLs. Resources can be anything that can be named, including objects, data, or services.

HTTP Methods:

RESTful services use HTTP methods to perform CRUD (Create, Read, Update, Delete) operations on resources. For example, GET retrieves data, POST creates data, PUT updates data, and DELETE removes data.

Statelessness:

REST services are stateless, meaning each request from a client to a server must contain all the information needed to understand and process the request. Servers do not store client state.

Representations:

Resources are represented in various formats, such as JSON or XML. Clients can request different representations of a resource using content negotiation.

Hypermedia:

RESTful APIs may include hypermedia links (HATEOAS - Hypermedia as the Engine of Application State) in responses, allowing clients to navigate the API by following links.

**Use Cases**:

Web APIs:

REST is commonly used to build web APIs for mobile and web applications. It provides a scalable and easy-to-use approach for exposing data and services.

Microservices:

RESTful APIs are often employed in microservices architectures, where services communicate with each other via HTTP to perform specific tasks.

Resource Management:

REST is useful for managing resources and performing CRUD operations on them, making it suitable for web applications and data manipulation.

Stateless Services:

REST's statelessness simplifies scaling and load balancing in distributed systems.

Interoperability:

RESTful services can be used across different platforms and technologies due to their reliance on standard HTTP methods and formats.

In summary, HTTP is the protocol that powers the World Wide Web, and REST is an architectural style for designing scalable and stateless web services. REST leverages HTTP methods and resources to create APIs that are widely used for web and mobile applications, microservices, and distributed systems. It provides a standardized and flexible approach to building web-based systems.

# WebSocket vs gRPC

WebSocket and gRPC are two distinct communication technologies used in system design, each with its own characteristics, implementation, and use cases:

## WebSocket:

**Definition**:

WebSocket is a communication protocol that provides full-duplex, bidirectional, real-time communication over a single, long-lived connection between a client and a server. It operates on top of the standard HTTP/HTTPS protocols.

**Implementation**:

Protocol:

WebSocket defines a protocol for handshake and data framing, allowing both the client and server to send messages to each other without the need for a request-response pattern.

Persistent Connection:

WebSocket connections are long-lived and remain open, enabling real-time data streaming and interaction. This is suitable for applications like chat, online gaming, live dashboards, and collaborative tools.

JavaScript API:

In web applications, WebSocket communication is typically implemented using JavaScript APIs that provide WebSocket support in web browsers.

Server Libraries:

Server-side implementations of WebSocket are available in various programming languages and frameworks, making it relatively easy to implement WebSocket support in a server application.

**Use Cases**:

**Real-Time Applications**: WebSocket is ideal for building real-time applications that require low-latency, bidirectional communication between clients and servers. Examples include instant messaging, live notifications, online gaming, and collaborative tools.

**Live Dashboards**: WebSocket can be used to update live dashboards with real-time data from various sources, providing users with up-to-the-minute information.

**Streaming Services**: It's suitable for streaming data services where clients need to receive continuous updates, such as stock market updates, weather feeds, or social media feeds.

## gRPC:

**Definition**:

gRPC is a high-performance, open-source framework for building remote procedure call (RPC) APIs. It uses Protocol Buffers (protobufs) as its interface definition language (IDL) and supports multiple programming languages.

**Implementation**:

IDL:

Developers define service methods and message structures in a .proto file using Protocol Buffers, which serves as a contract between clients and servers.

Code Generation:

gRPC generates client and server code in various programming languages from the .proto file, making it easy to develop and maintain consistent APIs across different platforms.

HTTP/2:

gRPC communicates over HTTP/2, a modern and efficient protocol that offers multiplexing, header compression, and other features to improve performance.

**Use Cases**:

Microservices:

gRPC is well-suited for building microservices architectures, where services communicate with each other using strongly typed, efficient RPCs.

Cross-Language Communication:

gRPC's support for multiple programming languages allows services written in different languages to communicate seamlessly. This is valuable in polyglot environments.

Efficiency:

gRPC's binary serialization and HTTP/2 support result in efficient communication, making it suitable for high-performance applications.

APIs and Backend Services:

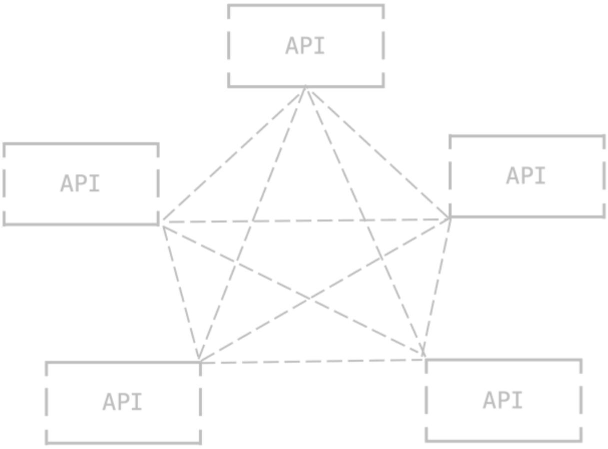
It can be used to build APIs for web applications and backend services, where efficient and type-safe communication is desired.

Streaming:

gRPC supports both unary and bidirectional streaming, allowing for efficient streaming of data between clients and servers.

In summary, WebSocket is a protocol for real-time, bidirectional communication commonly used in web applications, while gRPC is a framework for building efficient RPC-based APIs, often used in microservices architectures and cross-language communication. The choice between them depends on the specific requirements of your application, such as the need for real-time communication, type safety, and efficient data transfer.

## GraphQL



**GraphQL** is a query language for APIs and a runtime environment for executing those queries by utilizing a type system you define for your data. GraphQL is designed to be a more efficient, powerful, and flexible alternative to traditional RESTful APIs. Let's explore what GraphQL is, how it is typically implemented, and why it is useful in system design:

### Key Functions:

**Query Language**:

GraphQL allows clients to specify the shape and structure of the data they need by sending queries to the server. Clients can request only the data they require, avoiding over-fetching or under-fetching of data.

**Strongly Typed**:

GraphQL uses a type system to define the structure of your data. You specify the types, their relationships, and the available queries and mutations in a schema.

**Single Endpoint**:

Unlike REST, which typically exposes multiple endpoints for different resources, GraphQL uses a single endpoint for all data queries and mutations.

**Real-Time Data**:

GraphQL supports real-time data using subscriptions. Clients can subscribe to specific events and receive updates when relevant data changes.

## Implementation:

**Schema Definition**:

In GraphQL, you start by defining a schema. The schema specifies the types available, their relationships, and the queries and mutations that can be performed. This schema serves as a contract between clients and servers.

**Resolvers**:

For each field in your schema, you provide resolver functions. Resolvers are responsible for fetching the data for their corresponding fields. They determine how to retrieve data from databases, APIs, or other sources.

**Query Execution**:

When a client sends a query, the GraphQL server parses the query, validates it against the schema, and executes the appropriate resolvers to fetch the requested data.

In summary, GraphQL is a powerful alternative to traditional RESTful APIs that offers greater efficiency, flexibility, and control over data fetching. It is particularly valuable in scenarios where clients have varying data requirements, need real-time updates, or require efficient data fetching in resource-constrained environments.

# SECURITY

## Data Encryption

Data encryption is a crucial aspect of system design, providing security for data both in transit (while it's being transmitted between systems) and at rest (when it's stored on storage devices).

Achieving data encryption in transit and at rest involves different mechanisms and considerations:

### Data Encryption in Transit:

Transport Layer Security (TLS) / Secure Sockets Layer (SSL):

> TLS/SSL protocols are used to encrypt data during transit over networks, such as the internet.
>
> TLS/SSL employs encryption algorithms to protect the confidentiality and integrity of data exchanged between a client and a server.
>
> It relies on digital certificates to authenticate the server and, optionally, the client.
>
> Implementing TLS/SSL is essential for securing sensitive information during communication, preventing eavesdropping, and protecting against man-in-the-middle attacks.

### Data Encryption at Rest:

Full Disk Encryption (FDE):

> FDE encrypts the entire storage device (e.g., hard drive, SSD) at the block level.
>
> When data is written to the disk, it's automatically encrypted, and when it's read, it's decrypted on-the-fly.

FDE ensures that if the physical storage device is stolen or compromised, the data remains inaccessible without the encryption key.

File-Level Encryption:

File-level encryption encrypts individual files or directories.

It provides more granular control over which files are encrypted and can be useful for encrypting specific sensitive files while leaving others unencrypted.

Database Encryption:

Database systems often offer encryption options to protect data at rest.

This can include encrypting the entire database, specific tables, or even individual columns containing sensitive data.

Database encryption helps safeguard data stored in databases from unauthorized access.

## Why Data Encryption is Important in System Design:

Data at Rest Protection:

Encrypting data at rest safeguards it in scenarios where physical storage devices may be lost, stolen, or improperly disposed of.

Data in Transit Protection:

Encryption in transit secures data as it travels over networks, preventing eavesdropping and interception by malicious actors.

Securing Sensitive Operations:

In many applications, sensitive operations (e.g., password storage, payment processing) involve handling confidential information. Encryption adds an extra layer of security to these processes.

Protection from Data Leaks:

In the event of a data breach, encrypted data is much less valuable to attackers because they cannot easily access the information without the decryption key.

# Checksums and Hashing

```
  Data           Checksum Calculation          Checksum
  +----+          +-----------------+         +-------+
  |  |    +----->|                 |    +--->|       |
  |Data|   |     |  Checksum       |   |   |Checksum|
  |  |   |      |  Calculation    |   |   |       |
  +---+   |     |                 |   |   +-------+
        |     +-----------------+   |
        |                      |
        |                      |
        |                      |
        |                      |
        |                      |
        |                      |
        |     +-----------------+   |
        +----->|                 |   |
              |  Checksum       |   |
              |  Verification   |----+
              |                 |
              +-----------------+
```

Checksums and hashing are fundamental concepts in computer science and system design, playing crucial roles in ensuring data integrity, security, and error detection. Here's how they work and why they are important:

## How Checksums Work:

**Properties:**

- A checksum is a fixed-size value calculated from data to detect errors or changes in that data.
- It involves summing or otherwise processing the bytes of data to produce a numeric value.
- The result is typically a smaller number that represents a summary or fingerprint of the data.
- Common checksum algorithms include CRC (Cyclic Redundancy Check) and Adler-32.

**Process:**

1. When data is transmitted or stored, a checksum is calculated and included with the data.
2. Upon receipt or retrieval, the receiver recalculates the checksum and compares it to the received checksum.
3. If the calculated and received checksums match, it suggests that the data has not been altered during transmission or storage.

## How Hashing Work:

- Hashing is a process of converting data (of arbitrary size) into a fixed-size string of characters or bytes, known as a hash value or hash code.
- The hash function takes an input (data) and applies a mathematical algorithm to produce a unique hash value.
- A well-designed hash function ensures that even a minor change in the input data results in a significantly different hash value.
- Hash functions are one-way; given the hash value, it is computationally infeasible to reverse-engineer the original input data.
- Hashes are commonly used in data structures like hash tables, for password storage, and in data integrity checks.

## Why Checksums and Hashing Are Important in System Design:

**Data Integrity**:

Checksums and hashing are essential for verifying the integrity of data. They help detect errors or unauthorized changes during data transmission or storage.

**Error Detection**:

In scenarios where data is prone to transmission errors (e.g., over networks), checksums can quickly identify corrupted data, allowing for retransmission or recovery.

**Data Verification**:

Hashing allows for data verification. When transmitting or storing data, a hash value is computed and transmitted or stored alongside the data. The recipient can verify the data's authenticity by rehashing and comparing it with the received hash.

**Security**:

Hashing is a fundamental component of password security. Storing password hashes instead of plaintext passwords enhances security by protecting user credentials.

**Efficient Data Retrieval**:

Hashing is used in data structures like hash tables for efficient data retrieval. It enables quick lookup and indexing of data based on a hash value.

**Digital Signatures**:

Hashing is used in digital signatures to ensure the authenticity and integrity of messages or documents. Hash values of the content are signed to prove that the data hasn't been altered.

**Data Deduplication**:

Hashing is used in data deduplication systems to identify and eliminate duplicate copies of data, saving storage space.

**Cryptography**:

Cryptographic protocols rely on hashing for functions such as message authentication codes (MACs) and digital certificates.

# Token-based authentication and sessions

Token-based authentication and sessions are two common approaches to managing user authentication in web applications. Here's a brief comparison of these two methods and how they are typically implemented:

## Token-Based Authentication:

```
  User                Server
    |                    |
    |   Authentication   |
    |------------------->|
    |                    |
    |                    |
    |   Token Issued     |
    |<-------------------|
    |                    |
    |                    |
    |  Token Sent        |
    |------------------->|
    |                    |
    |                    |
    |  Access Granted    |
    |<-------------------|
    |                    |
```

**Principle**:

Token-based authentication relies on the exchange of tokens between the client and server for user authentication.

Tokens are usually generated upon successful login and are used to prove the user's identity in subsequent requests.

**Stateless**:

Token-based authentication is typically stateless. The server doesn't store user session information.

Each request from the client must include the token for authentication.

**Scalability**:

Because it's stateless, token-based authentication can be more scalable in distributed systems. There's no need for session affinity or shared session storage.

**Implementation**:

The server generates a token (e.g., JSON Web Token or JWT) upon successful login and sends it to the client.

The client stores the token (often in local storage or cookies) and sends it with each subsequent request.

The server validates the token by checking its signature and claims. If valid, the user is authenticated.

## Session-Based Authentication:

```
User              Server
  |                  |
  |   Authentication   |
  |------------------->|
  |                  |
  |                  |
  |  Session Created    |
  |<-------------------|
  |                  |
  |                    |
  |  Session ID Sent     |
  |--------------------->|
  |                    |
  |                    |
  |  Session ID Checked  |
  |<---------------------|
  |                    |
  |                    |
  |   Access Granted     |
  |<---------------------|
  |                    |
```

**Principle**:

Session-based authentication relies on server-side storage of user session data.

Upon successful login, the server creates a session for the user and associates it with a session identifier (usually a cookie or URL parameter).

**Stateful**:

Sessions are stateful because the server stores session data on its side.

Each request from the client includes the session identifier.

**Implementation**:

After a user logs in, the server creates a session and sends a session identifier to the client, often in the form of a session cookie.

The client automatically includes the session identifier with each request.

The server looks up the session data associated with the identifier to authenticate the user.

## Key Differences:

**Storage**:

Token-based authentication stores user identity and claims in the token itself, while session-based authentication stores session data on the server.

**State**:

Token-based authentication is stateless, while session-based authentication is stateful.

**Scalability**:

Token-based authentication can be more scalable in distributed systems because there's no need for shared session storage or session affinity.

**Storage Location**:

Token-based authentication stores the token on the client (e.g., in cookies or local storage), while session-based authentication stores session data on the server.

**Logout Handling**:

In session-based authentication, server-side sessions can be invalidated easily for user logout. Token-based authentication requires additional handling to handle token invalidation and logout.

**Expiration**:

Tokens can be designed with expiration times, providing an additional level of security. Session expiration is typically managed by the server.

The choice between token-based authentication and session-based authentication depends on factors such as the specific use case, scalability requirements, and the desired trade-offs between statefulness and statelessness. Both methods can be secure when implemented correctly, and the choice should align with the project's architecture and goals.

# API Security

Securing a REST API is crucial to protect sensitive data and ensure that your application is not vulnerable to various security threats. Here are some best practices for securing a REST API:

**Use HTTPS (SSL/TLS)**:

Ensure that all communication between clients and the API is encrypted using HTTPS. This prevents eavesdropping and man-in-the-middle attacks.

**Authentication**:

Implement strong authentication mechanisms to verify the identity of clients. Common methods include API keys, tokens (e.g., OAuth), or username/password authentication.

Use strong and up-to-date authentication protocols (e.g., OAuth 2.0, OpenID Connect) for user authentication and authorization.

**Authorization**:

Implement fine-grained authorization to control what authenticated users can access and modify.

Use role-based access control (RBAC) or attribute-based access control (ABAC) to define permissions.

**Token-Based Authentication**:

When using token-based authentication, ensure that tokens have a limited lifespan (expiration time) and are securely stored and transmitted.

Implement token revocation mechanisms for logout or compromised tokens.

**API Rate Limiting**:

Enforce rate limits to prevent abuse or denial-of-service (DoS) attacks. Rate limiting should be based on the client's identity and usage patterns.

**Input Validation**:

Validate all input data to prevent injection attacks, such as SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF).

Implement input validation and sanitize user-generated content.

**Error Handling**:

Implement proper error handling to avoid exposing sensitive information in error messages.

Return standardized error responses with appropriate HTTP status codes.

**Security Headers**:

Use security headers in HTTP responses to enhance security. Common headers include Content Security Policy (CSP), X-Content-Type-Options, and X-Frame-Options.

**Cross-Origin Resource Sharing (CORS)**:

Configure CORS policies to control which domains are allowed to make requests to your API.

Limit cross-origin requests to only trusted domains.

**Data Encryption**:

Encrypt sensitive data at rest using encryption mechanisms provided by your database or storage solution.

Consider end-to-end encryption for data in transit between services.

**API Gateway**:

Consider using an API gateway for centralized security enforcement, including authentication, authorization, and traffic management.

**Security Standards and Frameworks**:

Follow security standards and frameworks like OWASP API Security Top Ten and OWASP Web Security Testing Guide to guide your security efforts.

Remember that security is an ongoing process, and it's essential to stay vigilant and adapt to evolving threats. Regularly review and update your security practices to mitigate new risks effectively.