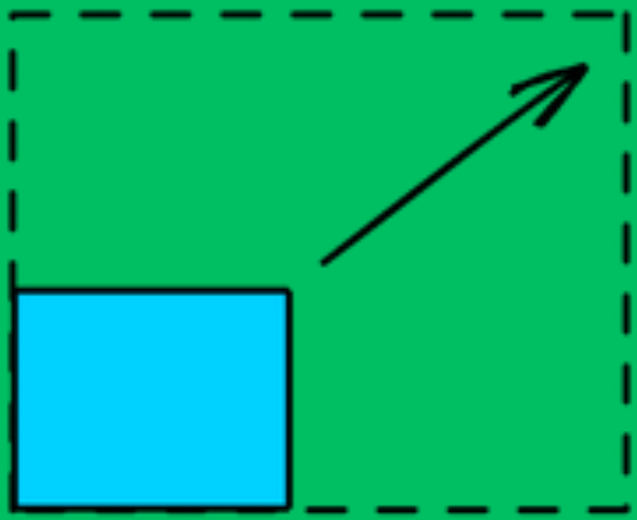
SYSTEM DESIGN
INTERVIEW HANDBOOK



75 pages guide to ace your next System Design Interview

Table of Contents

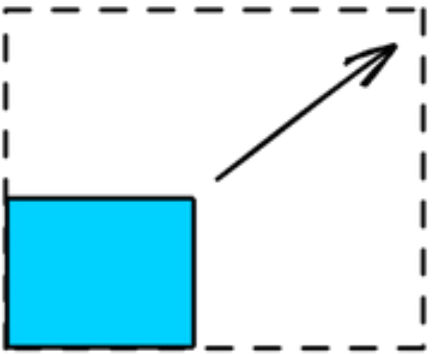
Fundamentals

Trade-offs

Architectural Patterns

SYSTEM DESIGN FUNDAMENTALS

1. Scalability



As a system grows, the performance starts to degrade unless we adapt it to deal with that growth.

Scalability is the property of a system to handle a growing amount of load by adding resources to the system.

A system that can continuously evolve to support a growing amount of work is **scalable**.

2. Availability

Availability refers to the proportion of time a system is operational and accessible when required.

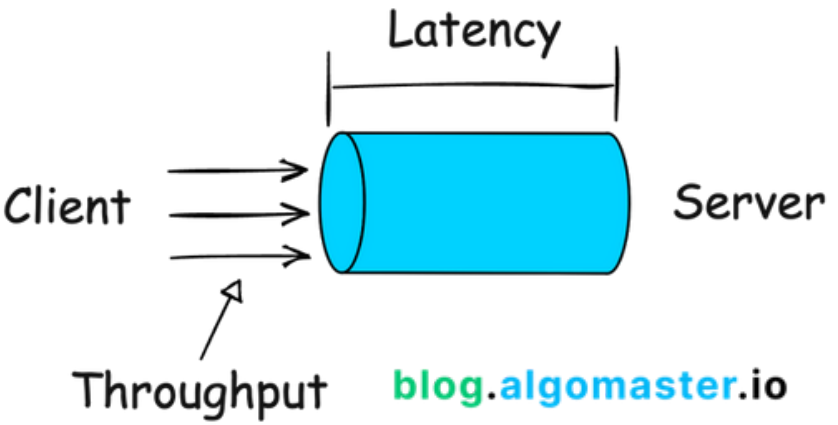
Availability = Uptime / (Uptime + Downtime)

Uptime: The period during which a system is functional and accessible.
Downtime: The period during which a system is unavailable due to failures, maintenance, or other issues.

Availability Tiers:

Availability %	Downtime per year	Commonly referred as
99%	3.65 days	"Two nines"
99.9%	8.76 hours	"Three nines"
99.99%	52.56 minutes	"Four nines"
99.999%	5.26 minutes	"Five nines"
99.9999%	31.5 seconds	"Six nines"

3. Latency vs Throughput



Latency

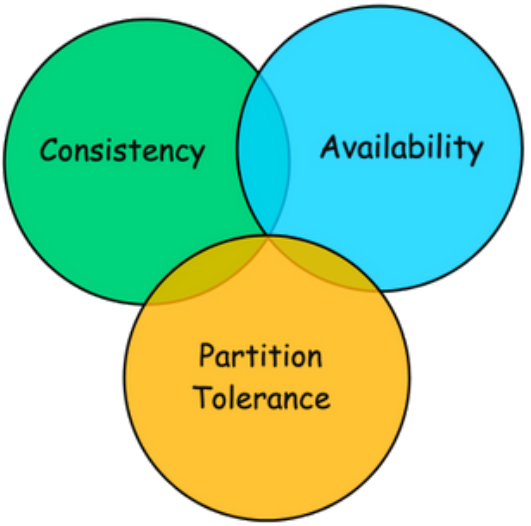
Latency refers to the time it takes for a single operation or request to complete.

- Low latency means faster response times and a more responsive system.
- High latency can lead to lag and poor user experience.

Throughput

Throughput measures the amount of work done or data processed in a given period of time.
It is typically expressed in terms of **requests per second (RPS)** or **transactions per second (TPS)**.

4. CAP Theorem

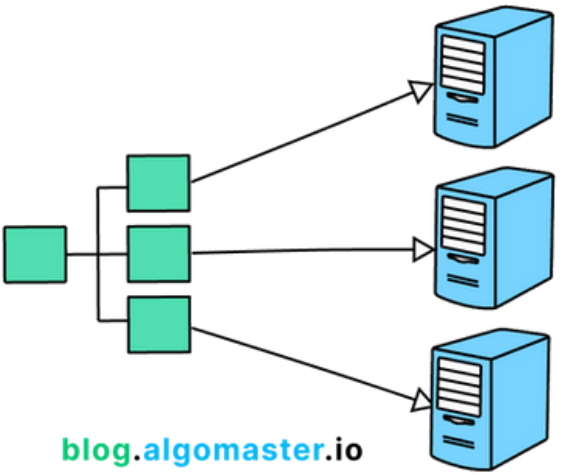


CAP stands for **Consistency**, **Availability**, and **Partition Tolerance**, and the theorem states that:

It is impossible for a distributed data store to simultaneously provide all three guarantees.

- **Consistency (C):** Every read receives the most recent write or an error.
- **Availability (A):** Every request (read or write) receives a non-error response, without guarantee that it contains the most recent write.
- **Partition Tolerance (P):** The system continues to operate despite an arbitrary number of messages being dropped (or delayed) by the network between nodes.

5. Load Balancers

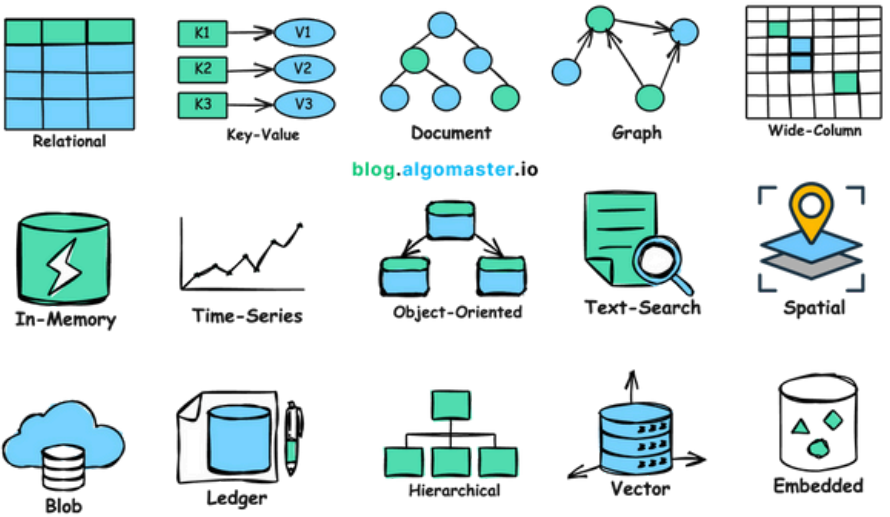


Load Balancers distribute incoming network traffic across multiple servers to ensure that no single server is overwhelmed.

Popular Load Balancing Algorithms:

1. **Round Robin:** Distributes requests evenly in circular order.
2. **Weighted Round Robin:** Distributes requests based on server capacity weights.
3. **Least Connections:** Sends requests to server with fewest active connections.
4. **Least Response Time:** Routes requests to server with fastest response.
5. **IP Hash:** Assigns requests based on hashed client IP address.

6. Databases

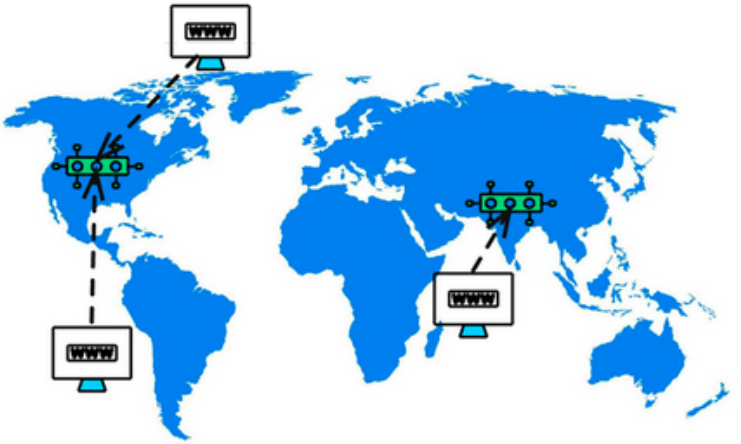


A database is an organized collection of structured or unstructured data that can be easily accessed, managed, and updated.

Types of Databases

1. Relational Databases (RDBMS)
2. NoSQL Databases
3. In-Memory Databases
4. Graph Databases
5. Time Series Databases
6. Spatial Databases

7. Content Delivery Network (CDN)

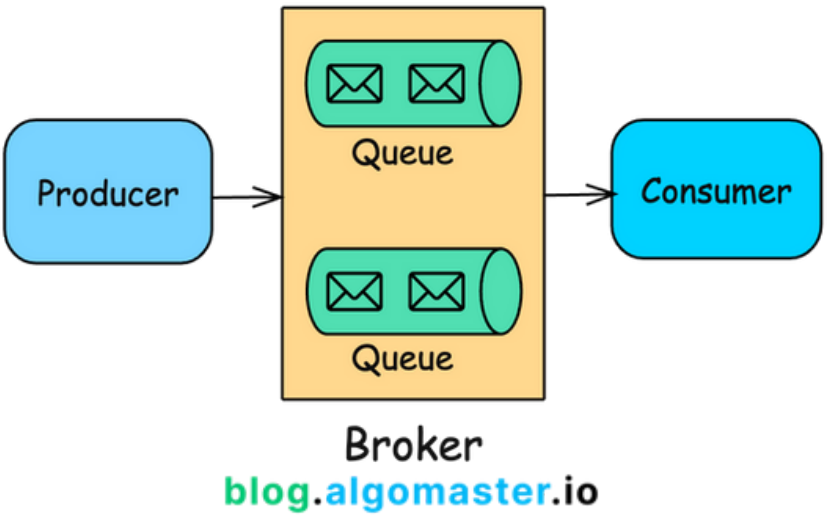


A **CDN** is a **geographically distributed** network of servers that work together to deliver web content (like HTML pages, JavaScript files, stylesheets, images, and videos) to users based on their geographic location.

The primary purpose of a CDN is to deliver content to end-users with high availability and performance by reducing the physical distance between the server and the user.

When a user requests content from a website, the CDN redirects the request to the nearest server in its network, reducing latency and improving load times.

8. Message Queues

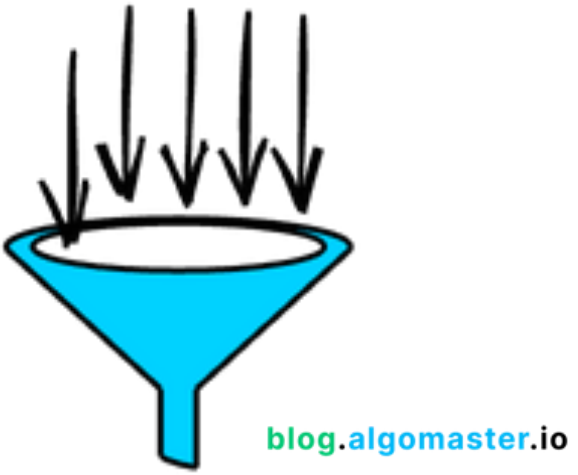


A message queue is a communication mechanism that enables different parts of a system to send and receive messages **asynchronously**.

Producers can send messages to the queue and move on to other tasks without waiting for consumers to process the messages.

Multiple consumers can pull messages from the queue, allowing work to be distributed and balanced across different consumers.

9. Rate Limiting

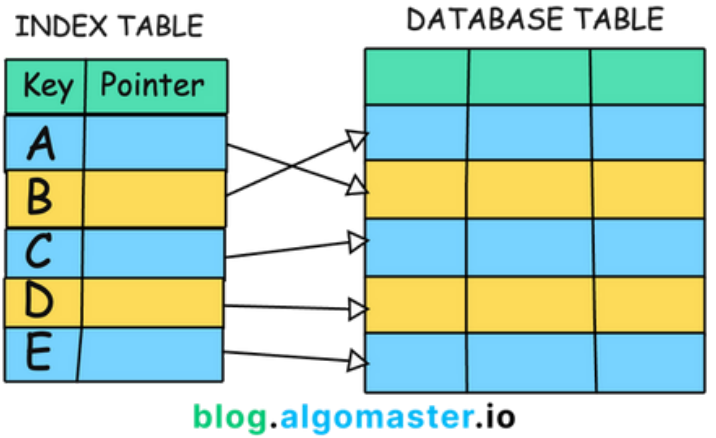


Rate limiting helps protects services from being overwhelmed by too many requests from a single user or client.

Rate Limiting Algorithms:

1. **Token Bucket:** Allows bursts traffic within overall rate limit.
2. **Leaky Bucket:** Smooths traffic flow at constant rate.
3. **Fixed Window Counter:** Limits requests in fixed time intervals.
4. **Sliding Window Log:** Tracks requests within rolling time window.
5. **Sliding Window Counter:** Smooths rate between adjacent fixed windows.

10. Database Indexes



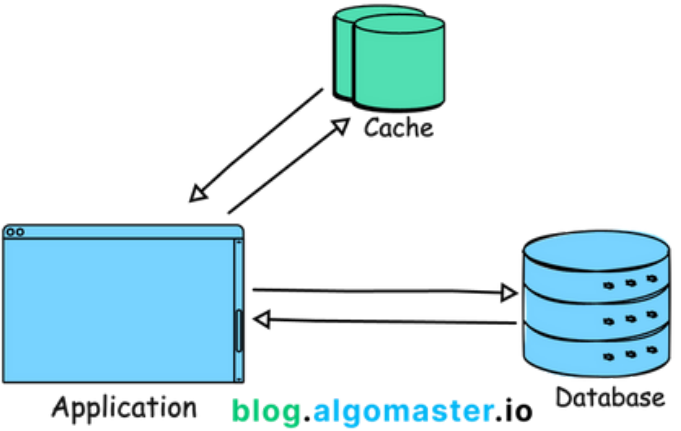
A **database index** is a super-efficient **lookup table** that allows a database to find data much faster.

It holds the indexed column values along with pointers to the corresponding rows in the table.

Without an index, the database might have to scan every single row in a massive table to find what you want – a painfully slow process.

But, with an index, the database can zero in on the exact location of the desired data using the index's pointers.

11. Caching



Caching is a technique used to temporarily store copies of data in **high-speed storage** layers to reduce the time taken to access data.

The primary goal of caching is to improve system performance by reducing latency, offloading the main data store, and providing faster data retrieval.

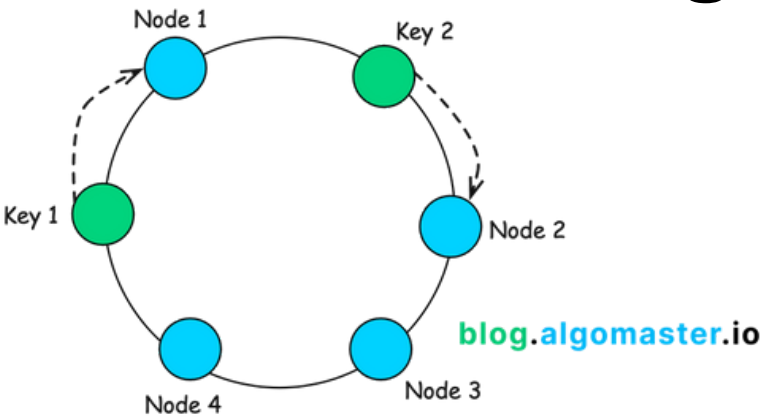
Caching Strategies:

1. **Read-Through Cache:** Automatically fetches and caches missing data from source.
2. **Write-Through Cache:** Writes data to cache and source simultaneously.
3. **Write-Back Cache:** Writes to cache first, updates source later.
4. **Cache-Aside:** Application manages data retrieval and cache population.

Caching Eviction Policies:

1. **Least Recently Used (LRU):** Removes the item that hasn't been accessed for the longest time.
2. **Least Frequently Used (LFU):** Discards items with the lowest access frequency over time.
3. **First In, First Out (FIFO):** Removes the oldest item, regardless of its usage frequency.
4. **Time-to-Live (TTL):** Automatically removes items after a predefined expiration time has passed.

12. Consistent Hashing



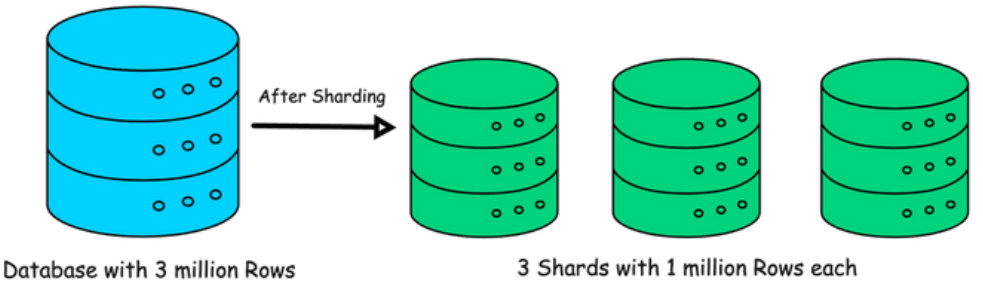
Consistent Hashing is a special kind of hashing technique that allows for **efficient distribution** of data across a cluster of nodes.

Consistent hashing ensures that only a small portion of the data needs to be reassigned when nodes are added or removed.

How Does it Work?

1. **Hash Space:** Imagine a fixed circular space or "ring" ranging from 0 to 2^n-1.
2. **Mapping Servers:** Each server is mapped to one or more points on this ring using a hash function.
3. **Mapping Data:** Each data item is also hashed onto the ring.
4. **Data Assignment:** A data item is stored on the first server encountered while moving clockwise on the ring from the item's position.

13. Database Sharding



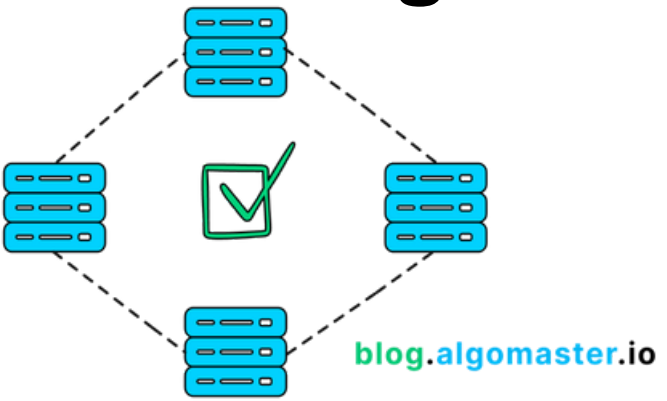
blog.algomaster.io

Database sharding is a horizontal scaling technique used to **split** a large database into smaller, independent pieces called shards.

These shards are then distributed across multiple servers or nodes, each responsible for handling a specific subset of the data.

By distributing the data across multiple nodes, sharding can significantly reduce the load on any single server, resulting in faster query execution and improved overall system performance.

14. Consensus Algorithms



In a distributed system, nodes need to work together to maintain a consistent state.

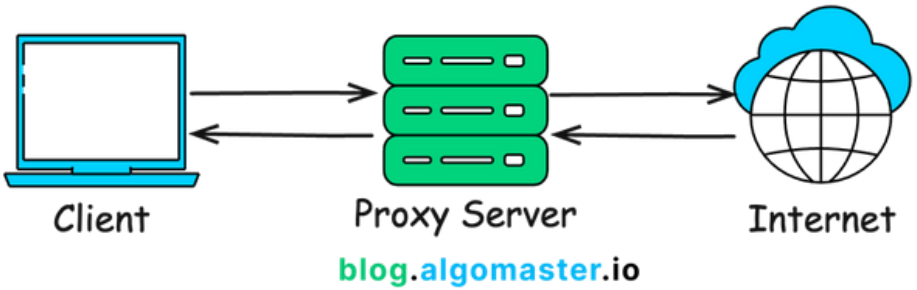
However, due to the inherent challenges like network latency, node failures, and asynchrony, achieving this consistency is not straightforward.

Consensus algorithms address these challenges by ensuring that all participating nodes agree on the same state or sequence of events, even when some nodes might fail or act maliciously.

Popular Consensus Algorithms

1. **Paxos:** Paxos works by electing a leader that proposes a value, which is then accepted by a majority of the nodes.
2. **Raft:** Raft works by designating one node as the leader to manage log replication and ensure consistency across the cluster.

15. Proxy Servers

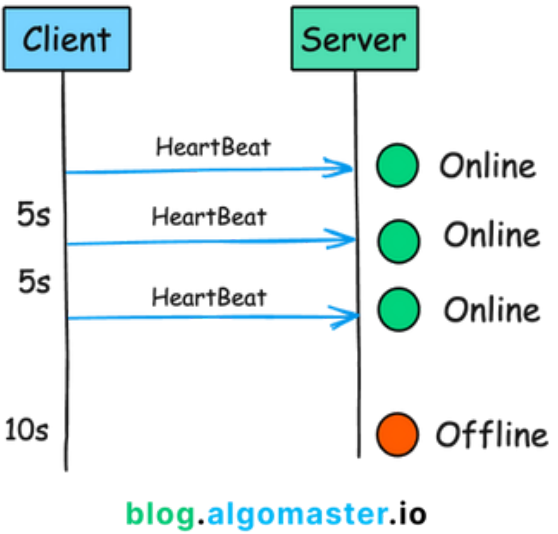


A **proxy server** acts as a **gateway** between you and the internet. It's an intermediary server separating end users from the websites they browse.

2 Common types of Proxy Servers:

1. **Forward Proxies:** Sits in front of a client and forwards requests to the internet on behalf of the client.
2. **Reverse Proxies:** Sits in front of a web server and forwards requests from clients to the server.

16. HeartBeats

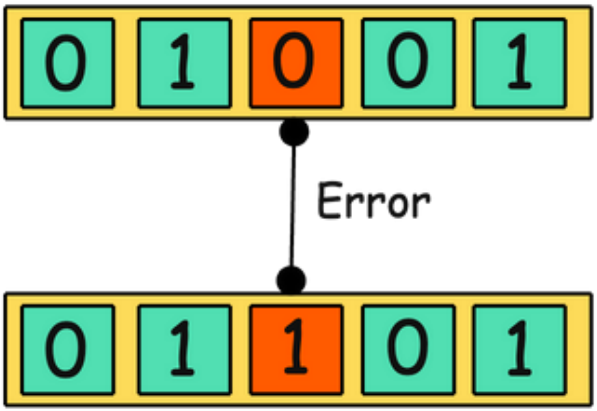


In distributed systems, a **heartbeat** is a **periodic message** sent from one component to another to monitor each other's health and status.

Without a heartbeat mechanism, it's hard to quickly detect failures in a distributed system, leading to:

- Delayed fault detection and recovery
- Increased downtime and errors
- Decreased overall system reliability

17. Checksums

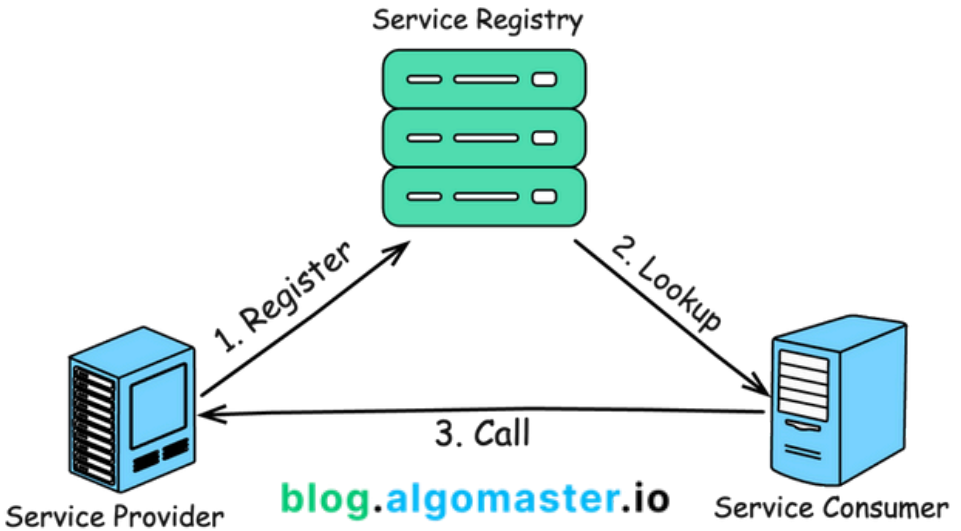


A **checksum** is a **unique fingerprint** attached to the data before it's transmitted.

When the data arrives at the recipient's end, the fingerprint is recalculated to ensure it matches the original one.

If the checksum of a piece of data matches the expected value, you can be confident that the data hasn't been modified or damaged.

18. Service Discovery



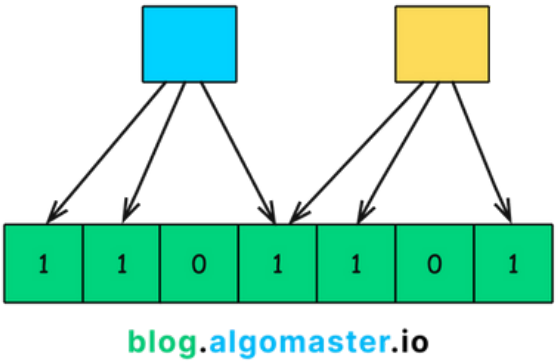
Service discovery is a mechanism that allows services in a distributed system to **find and communicate** with each other dynamically.

It hides the complex details of where services are located, so they can interact without knowing each other's exact network spots.

Service discovery registers and maintains a record of all your services in a **service registry.**

This service registry acts as a single source of truth that allows your services to query and communicate with each other.

19. Bloom Filters

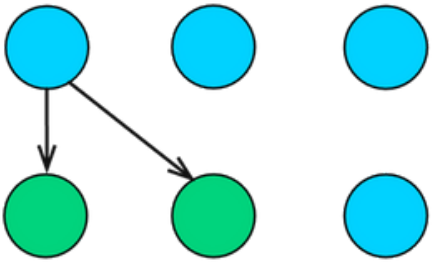


A **Bloom filter** is a **probabilistic data structure** that is primarily used to determine whether an element is definitely not in a set or possibly in the set.

How Does It Work?

1. **Setup:** Start with a bit array of m bits, all set to 0, and k different hash functions.
2. **Adding an element:** To add an element, feed it to each of the k hash functions to get k array positions. Set the bits at all these positions to 1.
3. **Querying:** To query for an element, feed it to each of the k hash functions to get k array positions. If any of the bits at these positions are 0, the element is definitely not in the set. If all are 1, then either the element is in the set, or we have a false positive.

20. Gossip Protocol



Gossip Protocol is a decentralized communication protocol used in distributed systems to spread information across all nodes.

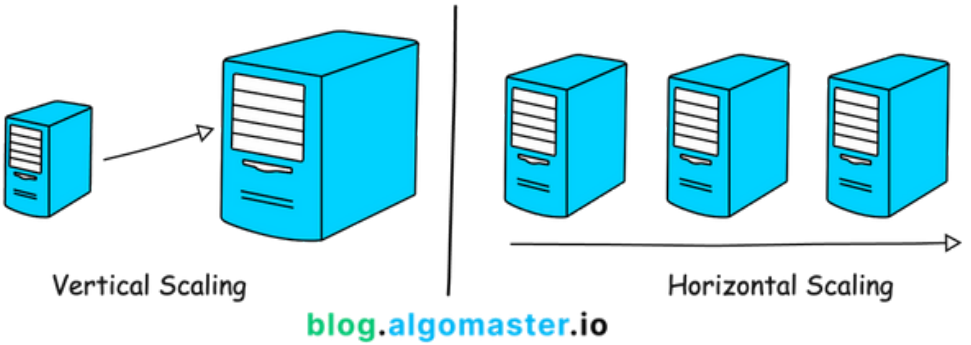
It is inspired by the way humans share news by word-of-mouth, where each person who learns the information shares it with others, leading to widespread dissemination.

How does it work?

1. **Initialization:** A node in the system starts with a piece of information, known as a "gossip."
2. **Gossip Exchange:** At regular intervals, each node randomly selects another node and shares its current gossip. The receiving node then merges the received gossip with its own.
3. **Propagation:** The process repeats, with each node spreading the gossip to others.
4. **Convergence:** Eventually, every node in the network will have received the gossip, ensuring that all nodes have consistent information.

SYSTEM DESIGN TRADE-OFFS

1. Vertical vs Horizontal Scaling



Vertical scaling involves boosting the power of an existing machine (eg.. CPU, RAM, Storage) to handle increased loads.

Scaling vertically is simpler but there's a physical limit to how much you can upgrade a single machine and it introduces a single point of failure.

Horizontal scaling involves adding more servers or nodes to the system to distribute the load across multiple machines.

Scaling horizontally allows for almost limitless scaling but brings complexity of managing distributed systems.

2. Strong vs Eventual Consistency

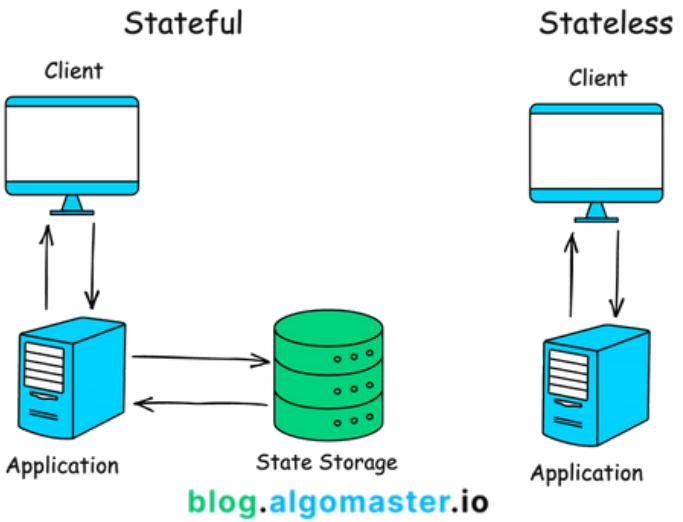
Strong consistency ensures that any read operation returns the most recent write for a given piece of data.

This means that once a write is acknowledged, all subsequent reads will reflect that write

Eventual consistency ensures that, given enough time, all nodes in the system will converge to the same value.

However, there are no guarantees about when this convergence will occur.

3. Stateful vs Stateless Design



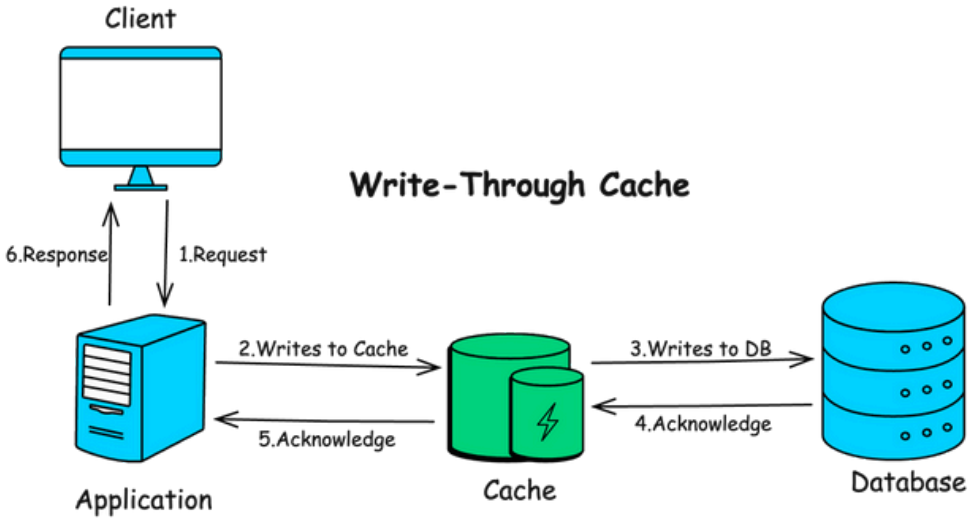
In a **stateful design**, the system remembers client data from one request to the next.

It maintains a record of the client's state, which can include session information, transaction details, or any other data relevant to the ongoing interaction.

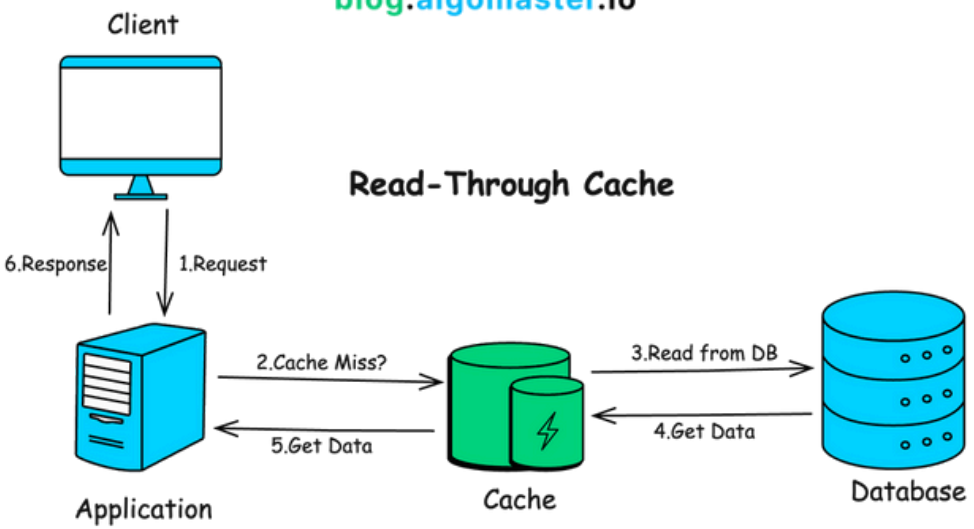
Stateless design treats each request as an independent transaction. The server does not store any information about the client's state between requests.

Each request must contain all the information necessary to understand and process it.

4. Read-Through vs Write-Through Cache



blog.algomaster.io



A **Read-Through cache** sits between your application and your data store.

When your application requests data, it first checks the cache.

If the data is found in the cache (a cache hit), it's returned to the application.

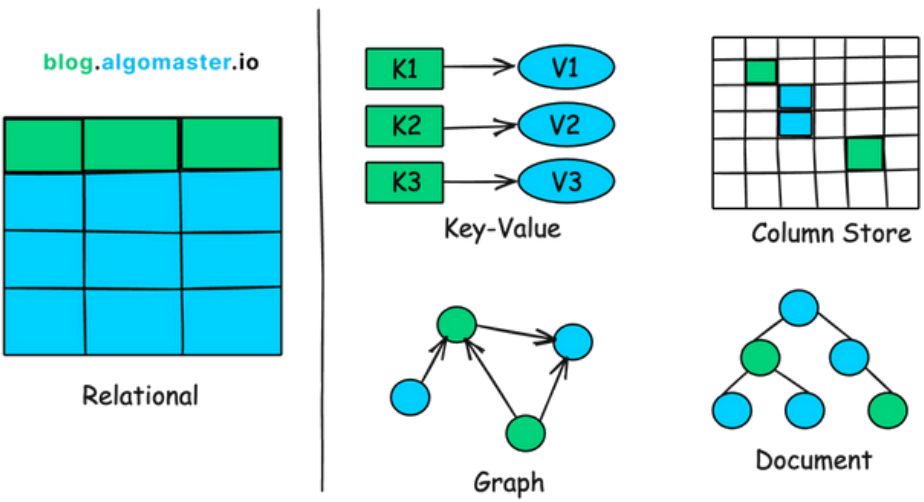
If the data is not in the cache (a cache miss), the cache itself is responsible for loading the data from the data store, caching it, and then returning it to the application.

In a **Write-Through cache** strategy, data is written into the cache and the corresponding database simultaneously.

Every write operation writes data to both the cache and the data store.

The write operation is only considered complete when both writes are successful.

5. SQL vs NoSQL



SQL databases use structured query language and have a predefined schema. They're ideal for:

- **Complex queries:** SQL is powerful for querying complex relationships between data.
- **ACID compliance:** Ensures data validity in high-stake transactions (e.g., financial systems).
- **Structured data:** When your data structure is unlikely to change.

Examples: MySQL, PostgreSQL, Oracle

NoSQL databases are more flexible and scalable. They're best for:

- Big Data: Can handle large volumes of structured and unstructured data.
- Rapid development: Schema-less nature allows for quicker iterations.
- Scalability: Easier to scale horizontally.

Examples: MongoDB, Cassandra, Redis

6. REST vs RPC

When designing APIs, two popular architectural styles often come into consideration: **REST (Representational State Transfer)** and **RPC (Remote Procedure Call)**. Both have their strengths and ideal use cases. Let's dive into their key differences to help you choose the right one for your project.

REST (Representational State Transfer)

REST is an architectural style that uses HTTP methods to interact with resources.

Key characteristics:

- Stateless: Each request contains all necessary information
- Resource-based: Uses URLs to represent resources
- Uses standard HTTP methods (GET, POST, PUT, DELETE)
- Typically returns data in JSON or XML format

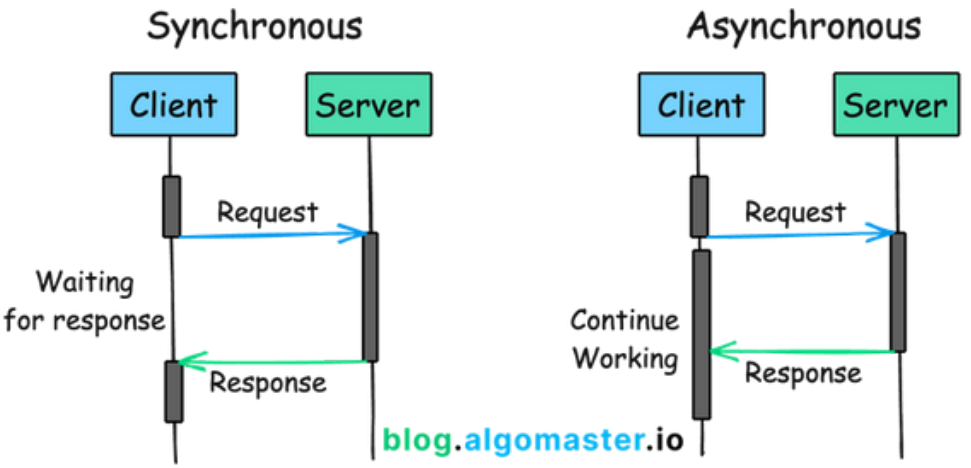
RPC (Remote Procedure Call)

RPC is a protocol that one program can use to request a service from a program located on another computer in a network.

Key characteristics:

- Action-based: Focuses on operations or actions
- Can use various protocols (HTTP, TCP, etc.)
- Often uses custom methods
- Typically returns custom data formats

7. Synchronous vs Asynchronous



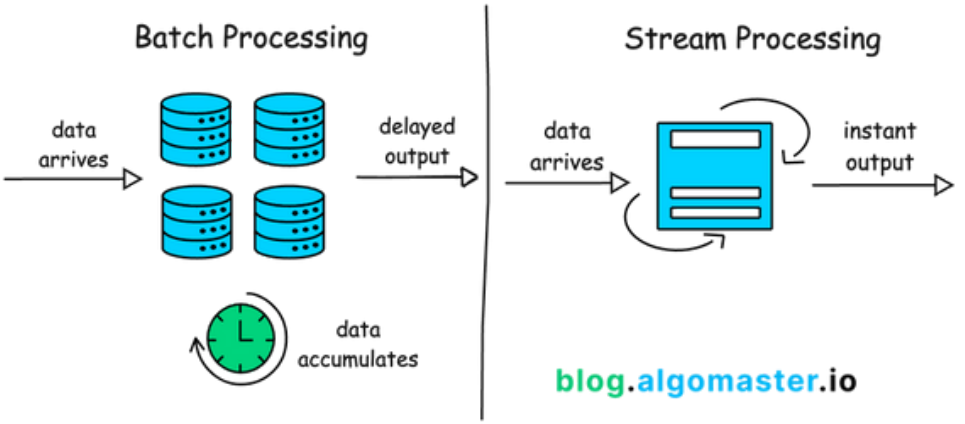
🔹 Synchronous Processing:

- Tasks are executed sequentially.
- Makes it easier to reason about code and handle dependencies.
- Used in scenarios where tasks must be completed in order like reading a file line by line.

🔹 Asynchronous Processing:

- Tasks are executed concurrently.
- Improves responsiveness and performance, especially in I/O-bound operations
- Used when you need to handle multiple tasks simultaneously without blocking the main thread. like background processing jobs.

8. Batch vs Stream Processing



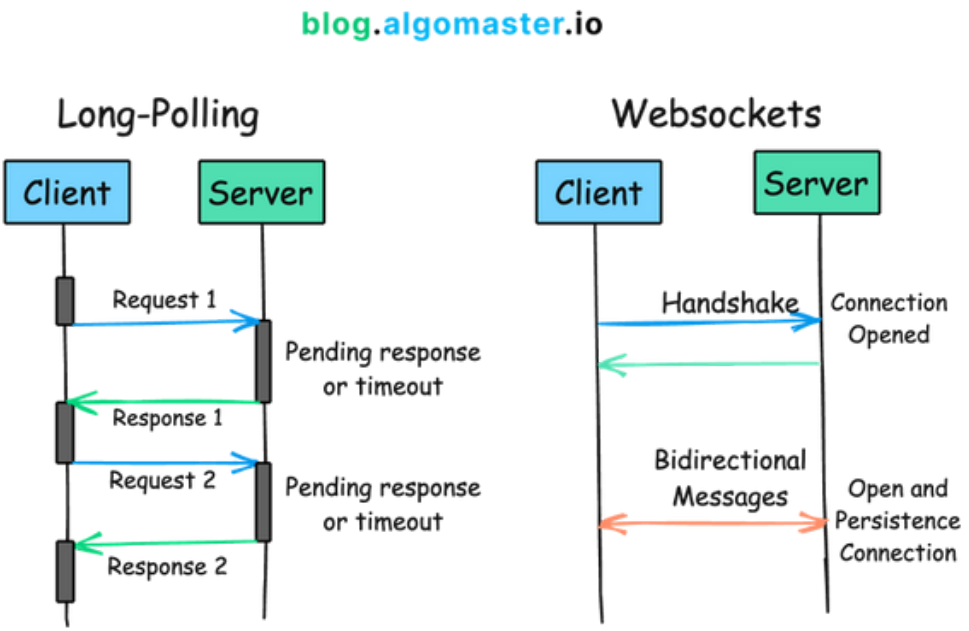
🔹 Batch Processing:

- Process large volumes of data at once, typically at scheduled intervals.
- Efficient for handling massive datasets, ideal for tasks like reporting or data warehousing.
- High Latency - results are available only after the entire batch is processed.
- Examples: ETL jobs, data aggregation, periodic backups.

🔹 Stream Processing:

- Process data in real-time as it arrives.
- Perfect for real-time analytics, monitoring, and alerting systems.
- Minimal latency since data is processed within milliseconds or seconds of arrival.
- Examples: Real-time fraud detection, live data feeds, IoT applications.

9. Long Polling vs WebSockets



In a **Long Polling** connection, the client repeatedly requests updates from the server at regular intervals.

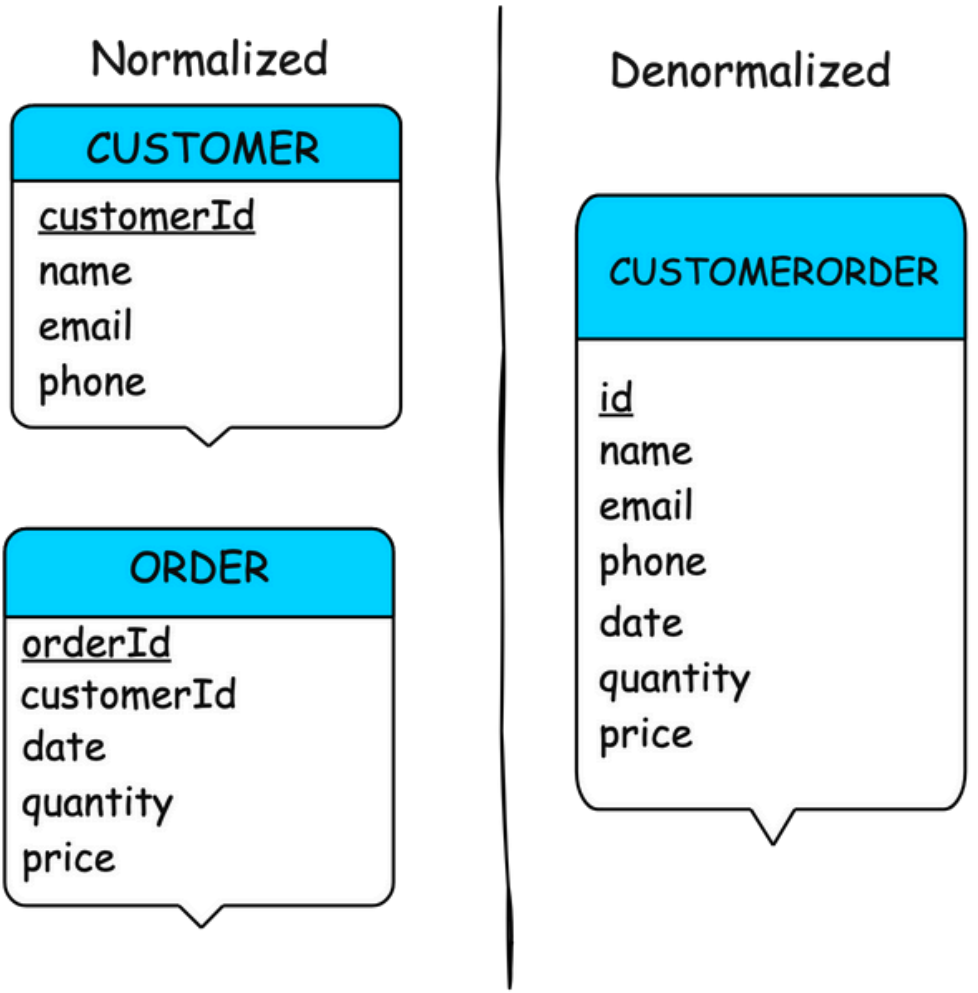
If the server has new data, it sends a response immediately; otherwise, it holds the connection until data is available.

This can lead to Increased latency and higher server load due to frequent requests, even when no data is available.

Websokcet establishes a persistent, full-duplex connection between the client and server, allowing real-time data exchange without the overhead of HTTP requests.

Unlike the traditional HTTP protocol, where the client sends a request to the server and waits for a response, WebSockets allow both the client and server to send messages to each other independently and continuously after the connection is established.

10. Normalization vs Denormalization



Normalization in database design involves splitting up data into related tables to ensure each piece of information is stored only once.

It aims to reduce redundancy and improve data integrity.

Example: A customer database can have two separate tables: one for customer details and another for orders, avoiding duplication of customer information for each order.

Denormalization is the process of combining data back into fewer tables to improve query performance.

This often means introducing redundancy (duplicate information) back into your database.

Example: A blog website can store the latest comments with the posts in the same table (denormalized) to speed up the display of post and comments, instead of storing them separately (normalized).

11. TCP vs UDP

When it comes to data transmission over the internet, two key protocols are at the forefront: TCP and UDP.

🔹 TCP (Transmission Control Protocol):

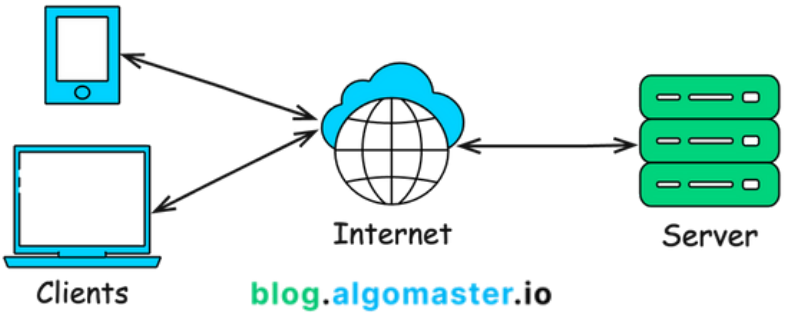
- **Reliable:** Ensures all data packets arrive in order and are error-free.
- **Connection-Oriented:** Establishes a connection before data transfer, making it ideal for tasks where accuracy is crucial (e.g., web browsing, file transfers).
- **Slower:** The overhead of managing connections and ensuring reliability can introduce latency.

🔹 UDP (User Datagram Protocol):

- **Faster:** Minimal overhead allows for quick data transfer, perfect for time-sensitive applications.
- **Connectionless:** No formal connection setup; data is sent without guarantees, making it ideal for real-time applications (e.g., video streaming, online gaming).
- **Unreliable:** No error-checking or ordering, so some data packets might be lost or arrive out of order.

ARCHITECTURAL PATTERNS

1. Client-Server Architecture

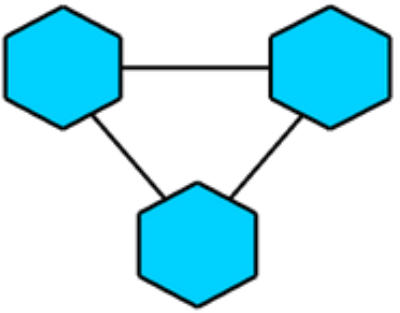


In this model, the system is divided into two main components: the client and the server.

- **Client:** The client is typically the user-facing part of the system, such as a web browser, mobile app, or desktop application. Clients send requests to the server and display the results to the user.

- **Server:** The server processes client requests, manages resources like databases, and sends the required data or services back to the client.

2. Microservices Architecture



blog.algomaster.io

Microservices architecture is an approach to designing a system as a collection of loosely coupled, independently deployable services.

Each microservice corresponds to a specific business function and communicates with other services via lightweight protocols, often HTTP/REST or messaging queues.

Services are small, focused on doing one thing well and each service has its own database to ensure loose coupling.

3. Serverless Architecture

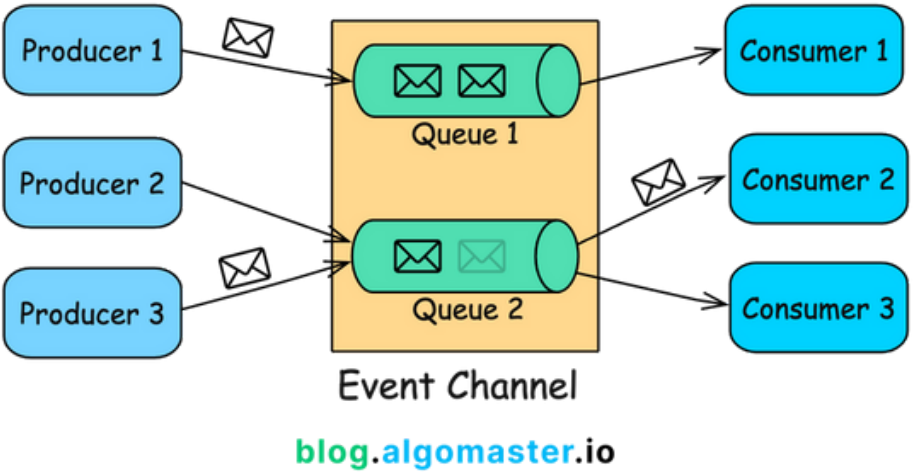
Serverless architecture abstracts away the underlying infrastructure, allowing developers to focus solely on writing code.

In a serverless model, the cloud provider automatically manages the infrastructure, scaling, and server maintenance.

Developers deploy functions that are triggered by events, and they are billed only for the compute time consumed.

Ideal for applications that react to events, such as processing files, triggering workflows, or handling real-time data streams.

4. Event-Driven Architecture

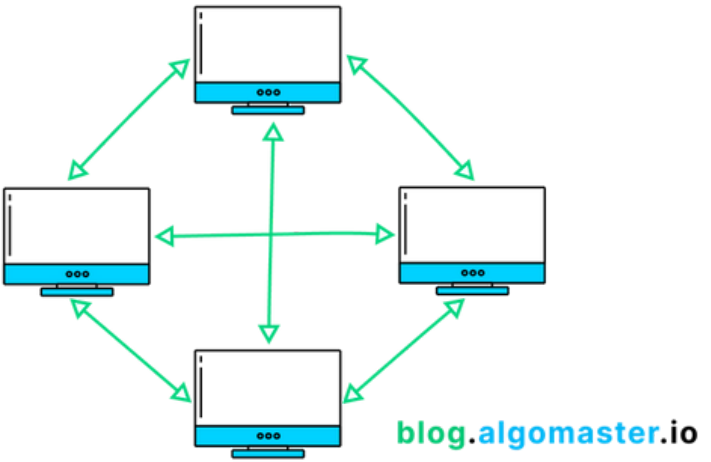


Event-Driven Architecture (EDA) is a design pattern in which the system responds to events, or changes in state, that are propagated throughout the system.

In EDA, components are decoupled and communicate through events, which are typically handled asynchronously.

Events can be processed in parallel by multiple consumers, allowing the system to scale efficiently.

5. Peer-to-Peer (P2P) Architecture



P2P architecture is a decentralized model where each node, or "peer," in the network has equal responsibilities and capabilities.

Unlike the client-server model, there is no central server; instead, each peer can act as both a client and a server, sharing resources and data directly with other peers.

P2P networks are known for their resilience and scalability since there is no central point of failure and system can scale easily as new peers join the network.

SYSTEM DESIGN INTERVIEW TEMPLATE

A step-by-step guide to System Design Interviews

Step 1. Clarify Requirements

Start by clarifying functional and non-functional requirements. Here are things to consider:

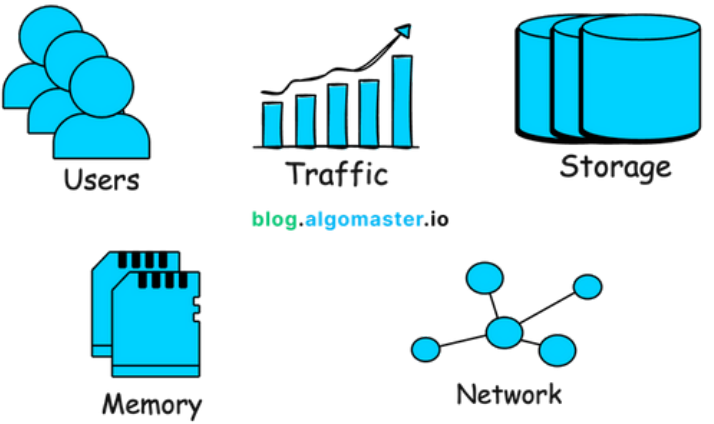
Functional Requirements:

- What are the core features that the system should support?
- Who are the users (eg.. customers, internal teams etc.)?
- How will users interact with the system (eg.. web, mobile app, API, etc.)?
- What are the key data types the system must handle (text, images, structured data, etc).
- Are there any external systems or third-party services the system needs to integrate with?

Non-Functional Requirements:

- Is the system read heavy or write heavy and what's the read-to-write ratio?
- Can the system have some downtime, or does it need to be highly available?
- Are there any specific latency requirements?
- How critical is data consistency?
- Should we rate limit the users to prevent abuse of the system?

Step 2. Capacity Estimation



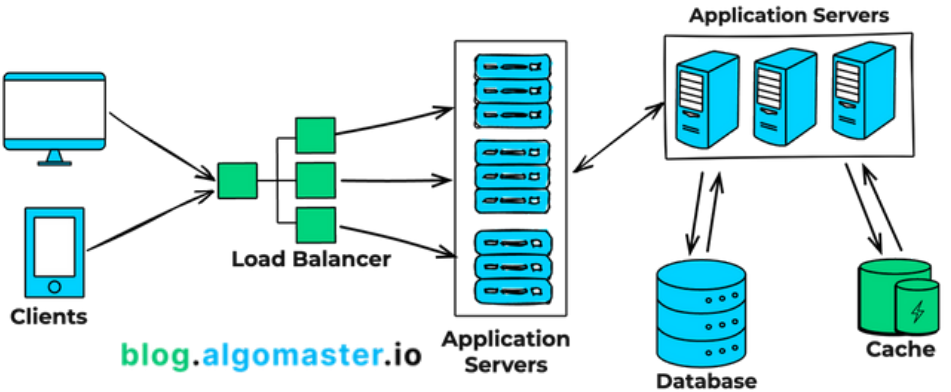
Estimate capacity to get an overall idea about how big a system you are going to design.

This can include things like:

- How many users are expected to use the system daily and monthly and maximum concurrent users during peak hours?
- Expected read/write requests per second.
- Amount of storage you would need to store all the data.
- How much memory you might need to store frequently accessed data in cache.
- Network bandwidth requirements based on the estimated traffic volume and data transfer sizes.

Note: Check with the interviewer if capacity estimation is necessary.

Step 3. High-Level Design



Sketch out a simple block diagram that outlines the major system components like:

1. **Clients:** User-facing interfaces (eg.. mobile, pc)
2. **Application Servers:** To process client requests.
3. **Load Balancers:** To distribute incoming traffic across multiple servers.
4. **Services:** Specialized components performing specific functions.
5. **Databases:** To store user information and metadata.
6. **Storage:** To store files, images or videos.
7. **Caching:** To improve latency and reduce load on the database.
8. **Message Queues:** If using asynchronous communication.
9. **External Services:** If integrating with third-party APIs (e.g., payment gateways).

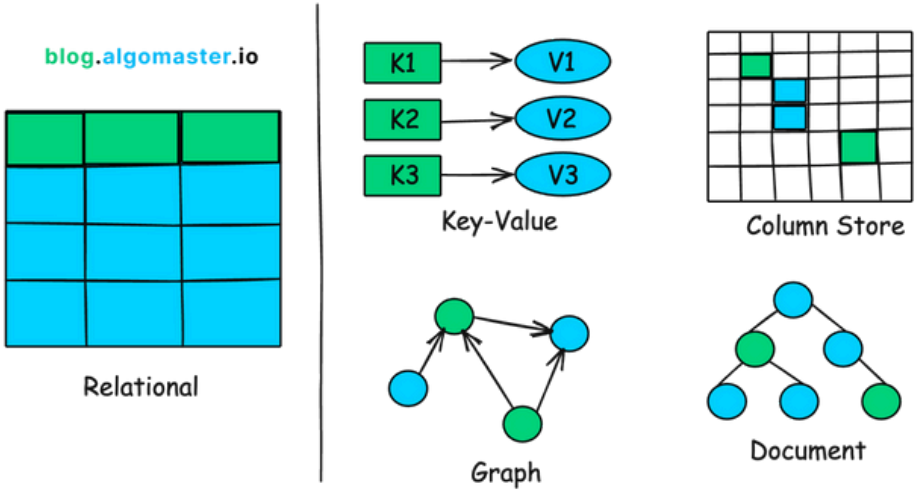
Step 4. Database Design

This steps involve modeling the data, choosing the right storage for the system, designing the database schema and optimizing the storage and retrieval of data based on the access patterns.

Data Modeling

- Identify the main **data entities** or objects that the system needs to store and manage (e.g., users, products, orders).
- Consider the **relationships** between these entities and how they interact with each other.
- Determine the **attributes** or properties associated with each entity (e.g., a user has an email, name, address).
- Identify any **unique identifiers** or primary keys for each entity.
- Consider **normalization** techniques to ensure data integrity and minimize redundancy.

Choose the Right Storage



- Evaluate the requirements and characteristics of the data to determine the most suitable database type.
- Consider factors such as data structure, scalability, performance, consistency, and query patterns.
- **Relational databases** (e.g., MySQL, PostgreSQL) are suitable for structured data with complex relationships and ACID properties.
- **NoSQL databases** (e.g., MongoDB, Cassandra) are suitable for unstructured or semi-structured data, high scalability, and eventual consistency.
- Consider using a combination of databases if different data subsets have distinct requirements.

Step 5. API Design

Define how different components of the system interact with each other and how external clients can access the system's functionality.

List down the APIs you want to expose to external clients based on the problem.

Select an appropriate API style based on the system's requirements and the clients' needs (eg.. RESTful, GraphQL, RPC).

Choose Communication Protocols:

- **HTTPS:** Commonly used for RESTful APIs and web-based communication.
- **WebSockets:** Useful for real-time, bidirectional communication between clients and servers (e.g., chat applications).
- **gRPC:** Efficient for inter-service communication in microservices architectures.
- **Messaging Protocols:** AMQP, MQTT for asynchronous messaging (often used with message queues).

Step 6. Dive Deep into Key Components

Your interviewer will likely want to focus on specific areas so pay attention and discuss those things in more detail.

It can differ based on the problem.

For example: if you are asked to design a url shortener, the interviewer will most likely want you to focus on the algorithm for generating short urls.

And, if the problem is about designing a chat application, you should talk about how the messages will be sent and received in real time.

Here are some more common areas of deep dives:

- **Databases:** How would you handle a massive increase in data volume? Discuss sharding (splitting data across multiple databases), replication (read/write replicas).
- **Application Servers:** How would you add more servers behind the load balancer for increased traffic?
- **Caching:** Where would you add caching to reduce latency and load on the database and how would you deal with cache invalidation?

Step 7. Address Key Concerns

This step involves identifying and addressing the core challenges that your system design is likely to encounter.

These challenges can range from scalability and performance to reliability, security, and cost concerns.

Addressing Scalability and Performance Concerns:

- Scale vertically (Scale-up) by increasing the capacity of individual resources (e.g., CPU, memory, storage).
- Scale horizontally (Scale-out) by adding more nodes and use load balancers to evenly distribute the traffic among the nodes.
- Implement caching to reduce the load on backend systems and improve response times.
- Optimize database queries using indexes.
- Denormalize data when necessary to reduce join operations.
- Use database partitioning and sharding to improve query performance.
- Utilize asynchronous programming models to handle concurrent requests efficiently.

Addressing Reliability

- Analyze the system architecture and identify potential single point of failures.
- Design redundancy into the system components (multiple load balancers, database replicas) to eliminate single points of failure.
- Consider geographical redundancy to protect against regional failures or disasters.
- Implement data replication strategies to ensure data availability and durability.
- Implement circuit breaker patterns to prevent cascading failures and protect the system from overload.
- Implement retry mechanisms with exponential backoff to handle temporary failures and prevent overwhelming the system during recovery.
- Implement comprehensive monitoring and alerting systems to detect failures, performance issues, and anomalies.

40
SYSTEM DESIGN INTERVIEW TIPS

1. Understand the **functional** and **non-functional** requirements before designing.

2. Clearly define the **use cases** and **constraints** of the system.

3. There is no perfect solution. It’s all about **tradeoffs**.

4. Assume everything can and will fail. Make it **fault tolerant.**

5. Keep it **simple**. Avoid over-engineering.

6. Design your system for **scalability** from the ground up.

7. Prefer **horizontal scaling** over vertical scaling for scalability.

8. Use **Load Balancers** to ensure high availability and distribute traffic.

9. Consider using **SQL** Databases for structured data and ACID transactions.

10. Opt for **NoSQL** Databases when dealing with unstructured data.

11. Consider using a **graph** database for highly connected data.

12. Use Database **Sharding** to scale SQL databases horizontally.

13. Use Database **Indexing** to optimize the read queries in databases.

14. Assume everything can and will fail. Make it **fault tolerant.**

15. Use **Rate Limiting** to prevent system from overload and DOS attacks.

16. Consider using **WebSockets** for real-time communication.

17. Use **Heartbeat** Mechanisms to detect failures.

18. Consider using a **message queue** for asynchronous communication.

19. Implement data **partitioning** and **sharding** for large datasets.

20. Consider **denormalizing** databases for read-heavy workloads.

21. Use **bloom filters** to check for an item in a large dataset quickly.

22. Use **CDNs** to reduce latency for a global user base.

23. Use **caching** to reduce load on the database and improve response times.

24. Use **write-through cache** for write-heavy applications.

25. Use **read-through cache** for read-heavy applications.

26. Use **object storage** like S3 for storing large datasets and media files.

27. Implement **Data Replication** and **Redundancy** to avoid single point of failure.

28. Implement **Autoscaling** to handle traffic spikes smoothly.

29. Use **Asynchronous processing** for background tasks.

30. Use **batch processing** for non-urgent tasks to optimize resources.

31. Make operations **idempotent** to simplify retry logic and error handling.

32. Consider using a **data lake** or data warehouse for analytics and reporting.

33. Implement comprehensive **logging and monitoring** to track the system's performance and health.

34. Implement **circuit breakers** to prevent a single failing service from bringing down the entire system.

35. Implement **chaos engineering** practices to test system resilience and find vulnerabilities.

36. Design for **statelessness** when possible to improve scalability and simplify architecture.

37. Use **failover mechanisms** to automatically switch to a redundant system when a failure is detected.

38. Distribute your system across different data centers to prevent localized failures.

39. Use **Time-To-Live (TTL)** values to automatically expire cached data and reduce staleness.

40. **Pre-populate** critical data in the cache to avoid cold starts.

10 MOST COMMON SYSTEM DESIGN INTERVIEW QUESTIONS

1. Design a URL Shortener like TinyURL

Functional Requirements:

- Generate a unique short URL for a given long URL
- Redirect the user to the original URL when the short URL is accessed
- Allow users to customize their short URLs (optional)
- Support link expiration where URLs are no longer accessible after a certain period
- Provide analytics on link usage (optional)

Non-Functional Requirements:

- **High availability:** The service should be up 99.9% of the time.
- **Low latency:** Url shortening and redirects should happen in milliseconds.
- **Scalability:** The system should handle millions of requests per day.
- **Durability:** Shortened URLs should work for years.
- **Security** to prevent malicious use, such as phishing.

2. Design a Chat Application like Whatsapp

Functional Requirements:

- Support one-on-one and group conversations between users.
- Keep track of online/offline status of users.
- Provide message delivery statuses (sent, delivered, read).
- Support multimedia messages (images, videos, voice notes, documents).
- Push notifications for new messages, calls, and mentions (optional)

Non-Functional Requirements:

- **Real-time** message delivery with minimal latency.
- The system should handle millions of concurrent users.
- The system should be **highly available**. However, the availability can be compromised in the interest of consistency.
- **Durability** (messages shouldn’t get lost)

3. Design a social media platform like Instagram

Functional Requirements:

- Users can upload and share images and videos.
- Users can like, comment, and share posts.
- Users can follow/unfollow other users.
- Generate and display news feed for users showing posts from people the user follows.
- Support for tagging other users in posts and comments.

Non-Functional Requirements:

- **High availability:** The service should be up 99.9% of the time.
- **Low latency** for news feed generation.
- **High Scalability:** The platform should handle millions of concurrent users.
- **High Durability:** User's uploaded photos and videos should't get lost.
- **Eventual Consistency:** If a user doesn't see a photo for sometime, it should be fine.

4. Design a video streaming service like YouTube

Functional Requirements:

- Users can upload videos.
- Support for adding video titles, descriptions, tags, and thumbnails.
- Users can stream videos on-demand.
- Search functionality to find videos, channels, and playlists based on keywords.
- Users can like, dislike, comment on, and share videos.
- Service should record view count of videos.

Non-Functional Requirements:

- **High availability** (e.g., 99.99% uptime) to ensure the service is accessible at all times.
- **Low latency:** Video streaming should be real-time without lag
- **High Scalability:** The service should be able to scale horizontally to accommodate increasing numbers of users and video content.
- **High Durability:** Uploaded videos shouldn’t get lost)

5. Design an E-commerce Platform like Amazon

Functional Requirements:

- Allow sellers to list products with details like title, description, price, images, and specifications.
- Users can add products to a shopping cart and wishlist.
- Users can search for products, categories, and brands based on keywords
- Users can place orders for one or multiple products.
- Users can rate and review products they have purchased.

Non-Functional Requirements:

- **High Scalability:** The platform should handle millions of users, products, and transactions simultaneously.
- **High Availability:** The service should be up 99.9% of the time.
- **Low latency** for page load times, search queries, and checkout processes.
- **High Durability:** All critical data (user data, product listings, orders) is stored with high durability.

6. Design a Ride-Sharing Service like Uber

Functional Requirements:

- Riders can see all the nearby available drivers.
- Riders can book a ride by specifying a pickup location, drop-off location.
- Match riders with nearby available drivers in real-time.
- Real-time estimation of ride cost and time of arrival based on distance, traffic, and demand.
- Real-time tracking of the ride both for rider and driver.

Non-Functional Requirements:

- **Scalability:** The system should scale to handle millions of users and rides simultaneously.
- Ensure **low latency** for real-time matching, GPS tracking, and ride requests.
- Ensure **high availability** with minimal downtime (e.g., 99.9% uptime).
- **Consistency:** The riders and drivers should have a consistent view of the system.

7. Design a File Storage Service like Google Drive

Functional Requirements:

- Users should be able to upload files of various types and sizes.
- Users should be able to download files on-demand.
- Users should be able to share files and folders with other users via links or email invitations.
- Allow users to search for files and folders by name, type, content, or metadata.
- Support synchronization of files across multiple devices.

Non-Functional Requirements:

- **Scalability:** The system should be able to handle millions of users and billions of files.
- Ensure **low-latency** file uploads, downloads, and search operations.
- **Availability:** The service should be highly available, with minimal downtime.
- Ensure consistency across all user devices during file synchronization.
- **Durability:** User's files should not be lost.

8. Design a Web Crawler

Functional Requirements:

- The system should be able to fetch URLs from the web efficiently.
- Handle different content types (e.g., text, images, multimedia).
- Prioritize URLs based on specific criteria (e.g., importance, freshness).
- Store crawled data efficiently in a database or file system.

Non-Functional Requirements:

- **Scalability:** The system should scale to handle millions or billions of web pages.
- **Minimize latency** in fetching and processing web pages.
- **Optimize throughput** to maximize the number of pages crawled per unit of time.

9. Design a Notification System

Functional Requirements:

- Support for various notification channels such as email, SMS, push notifications (mobile/web), and in-app notifications.
- Support for different types of notifications, such as transactional, promotional, and informational.
- Ability to schedule notifications for future delivery.
- Ability to send notifications in bulk, especially for campaigns or mass updates.
- Automatic retry mechanisms for failed notification deliveries.

Non-Functional Requirements:

- **High Scalability:** The system should handle millions of notifications per minute, especially during peak times.
- **High availability** (e.g., 99.99% uptime) to ensure notifications are sent without interruption.
- **Low latency** for sending notifications, especially for real-time and high-priority notifications.
- **Reliability:** The system should ensure reliable delivery of notifications across all supported channels.

10. Design a Logging and Monitoring System

Functional Requirements:

- Collect logs from various sources such as applications, servers, databases, and microservices.
- Support for multiple log formats (e.g., JSON, plaintext, XML).
- Archive old logs to cost-effective storage (e.g., cloud-based cold storage).
- Provide powerful querying capabilities to filter and search logs based on time range, log level, source, and other attributes.
- Set up alerts based on specific log patterns, thresholds, or anomalies.

Non-Functional Requirements:

- **Scalability:** The system should scale horizontally to handle increasing volumes of logs, metrics, and monitoring data.
- **Low-latency** log ingestion and processing to ensure real-time monitoring.
- **High availability** with minimal downtime (e.g., 99.99% uptime) to ensure continuous monitoring.
- **Durability:** Ensure that log data is stored with high durability, with replication across multiple nodes or data centers.