System Design_101

software engineers struggle with system design interviews (SDIs) primarily because of reasons:

- The unstructured nature of SDIs, where they are asked to work on an open-ended design problem that doesn't have a standard answer.
- Their lack of experience in developing large scale systems.

Whenever we are designing a large system, we need to consider a few things:

1. What are the different architectural pieces that can be used?
2. How do these pieces work with each other?
3. How can we best utilize these pieces: what are the right tradeoffs?

Step by step approach

1. Requirements Clarification

- It is always a good idea to ask questions about the exact scope of the problem we are solving. Design questions are mostly open-ended, and they don't have ONE correct answer, that's why clarifying ambiguities early in the interview becomes critical.

example of designing a Twitter-like service. Here are some questions for designing Twitter that should be answered before moving on to the next steps:

• Will users of our service be able to post tweets and follow other people?
• Should we also design to create and display the user's timeline?
• Will tweets contain photos and videos?
• Are we focusing on the backend only or are we developing the front-end too?
• Will users be able to search tweets?
• Do we need to display hot trending topics?
• Will there be any push notification for new (or important) tweets?

2. System interface definition

Define what APIs are expected from the system.
examples for Twitter-like service will be:

postTweet(user_id, tweet_data, tweet_location, user_location, timestamp, …)

generateTimeline(user_id, current_time, user_location, …)

markTweetFavorite(user_id, tweet_id, timestamp, …)

3. Back of the envelope estimation

estimate the scale of the system we're going to design. This will also help later when we will be focusing on scaling, partitioning, load balancing and caching.

- What scale is expected from the system (e.g., number of new tweets, number of tweet views, number of timeline generations per sec., etc.)?

• How much storage will we need? We will have different numbers if users can have photos and videos in their tweets.

• What network bandwidth usage are we expecting? This will be crucial in deciding how we will manage traffic and balance load between servers.

4. Defining data model

Defining the data model early will clarify how data will flow among different components of the system. Later, it will guide towards data partitioning and management. identify various entities of the system, how they will interact with each other, and different aspect of data management like storage, transportation, encryption, etc.

- User: UserID, Name, Email, DoB, CreationData, LastLogin, etc.
- Tweet: TweetID, Content, TweetLocation, NumberOfLikes, TimeStamp, etc.
- UserFollowor: UserdID1, UserID2
- FavoriteTweets: UserID, TweetID, TimeStamp

5. High level design

a block diagram with 5-6 boxes representing the core components of our system

6. Detailed Design

Dig deeper into two or three components; interviewer's feedback should always guide us what parts of the system need further discussion. We should be able to present different approaches, their pros and cons, and explain why we will prefer one approach on the other.

- Since we will be storing a massive amount of data, how should we partition our data to distribute it to multiple databases? Should we try to store all the data of a user on the same database? What issue could it cause?
- How will we handle hot users who tweet a lot or follow lots of people?
- Since users' timeline will contain the most recent (and relevant) tweets, should we try to store our data in such a way that is optimized for scanning the latest tweets?
- How much and at which layer should we introduce cache to speed things up?

7. Identifying and resolving bottlenecks

identify as many bottlenecks as possible and different approaches to mitigate them.

Data Partitioning

Data partitioning (also known as **data sharding**) refers to the process of splitting a large dataset into smaller, more manageable pieces, called **partitions** or **shards**, and distributing these pieces across multiple storage systems or servers. The primary goal is to improve performance, scalability, and availability of the system, especially in distributed environments.

Range-based Partitioning: Data is partitioned based on a predefined range of values. For example, records with timestamps between 01-01-2023 and 01-01-2024 might go into one partition, while the next year's data goes into another.

Hash-based Partitioning: A hash function is applied to a key (e.g., user ID, product ID) to determine which partition the data should belong to. This approach ensures an even distribution of data across partitions but can be difficult to manage when scaling.

List-based Partitioning: Data is partitioned based on a list of values. For instance, all records for users from the United States could go into one partition, and users from Europe into another.

a. Horizontal Partitioning (Sharding)

- **Definition**: In horizontal partitioning, data is divided by rows and stored in separate tables or servers. This is also known as **range-based sharding** when data is split based on a range of values (e.g., ZIP codes).
- **Example**:
  - Locations with ZIP codes less than 10000 are stored in one table.
  - Locations with ZIP codes greater than 10000 are stored in another table.
- **Key Issues**:
  - **Unbalanced load**: If the range chosen for partitioning is not well-distributed, it can lead to uneven load across servers. For example, in a ZIP code-based partition, densely populated areas (like Manhattan) will have more entries than less populated areas, leading to unbalanced server loads.

b. Vertical Partitioning

- **Definition**: In vertical partitioning, data is divided based on features or attributes and stored in separate tables or servers. Each table holds data related to a specific feature.
- **Example**:
  - User profile information might be stored in one database server.
  - Friend lists on a second server.
  - Photos and photo metadata on a third server.
- **Key Advantages**:
  - **Low impact**: It is relatively easy to implement and has minimal impact on application logic.
- **Key Issues**:
  - **Scalability**: As the system grows, the amount of data for each feature might increase beyond the capacity of a single server. For instance, storing metadata for billions of photos on a single server could become a bottleneck.

Common Problems of Sharding

Sharding introduces additional complexities and constraints to database operations due to the distribution of data across multiple servers. Below are some of the key problems associated with sharding:

a. Joins and Denormalization

- **Problem**: Performing joins on a sharded database can be inefficient because the data required for the join may be distributed across multiple servers. This can significantly degrade performance.
- **Solution**: A common workaround is **denormalization**, where data that would have been joined across multiple tables is stored in a single table. However, this introduces challenges like **data inconsistency** because maintaining consistency across denormalized tables can be difficult.

b. Referential Integrity

- **Problem**: Enforcing data integrity constraints like **foreign keys** in a sharded database is challenging. Most relational databases do not support foreign key constraints across databases on different servers.
- **Solution**: Since foreign key enforcement is not feasible across shards, applications must handle referential integrity at the application level. This often involves running periodic SQL jobs to clean up **dangling references** (data inconsistencies caused by missing foreign key relationships).

c. Rebalancing

- **Problem**: Over time, data distribution across shards may become uneven, leading to problems like:
    1. **Uneven Data Distribution**: Some shards may become overloaded because certain data (e.g., a popular ZIP code) does not fit evenly across partitions.
    2. **Overloaded Shards**: Some shards may experience higher loads due to the nature of the data (e.g., a shard dedicated to storing user photos may experience high read/write traffic).
- **Solution**: Rebalancing requires **changing the partitioning scheme** and moving data across servers. This is complex and typically requires **downtime**.

Cache

Cache is a high-speed data storage layer that stores a subset of data, typically data that is frequently accessed, to speed up data retrieval. Caching is used in various computing systems to reduce latency, avoid repeated database queries, and improve the overall performance of applications.

A cache is essentially a temporary storage location that holds data that is expensive or time-consuming to retrieve from a primary data store.

Advantages

- ***Faster Data Access**:
- ***Reduced Latency:**
- ***Lower Load on Backend Systems**:

- ***Scalability:**

How Does Caching Work?

1. **Data Request**: When a system or user requests data, the cache is checked to see if it already contains the data.
    - **Cache Hit**: If the data is found in the cache (called a "cache hit"), it is returned immediately, avoiding the need to retrieve it from the slower data source.
    - **Cache Miss**: If the data is not found in the cache (a "cache miss"), it is fetched from the backend (e.g., a database, file system, or API) and placed into the cache for future access.
2. **Eviction**: Caches have limited space, so when the cache is full, older or less frequently accessed data must be evicted (removed) to make room for new data. This process is governed by **eviction policies**.

Types of Caches:

- **Memory Cache**: This is the most common type, where data is cached in **RAM**. Examples include **Memcached** or **Redis**. Memory caches are very fast, as accessing data from RAM is much faster than reading it from a disk or database.
- **Disk Cache**: This type of cache stores data on disk. It’s slower than memory cache but still faster than repeatedly querying a database. Disk caches are often used when memory space is limited.
- **Web Cache**: Used to store HTTP responses (web pages, images, etc.) to speed up web browsing. A common example is **browser cache**—the web pages, images, and scripts that your browser stores locally to speed up repeated visits to the same websites.
- **Database Cache**: A cache used to store results of database queries. Systems like **MySQL Query Cache** or **Database Connection Pooling** use caching to reduce database load and improve query performance.
- **Application Cache**: Often implemented within software frameworks to store and retrieve application data that is expensive to calculate or fetch, such as user session data, temporary configurations, or precomputed results.

6. Where is Caching Used?

- **Web Servers**: To cache static resources (e.g., images, CSS, JavaScript) and dynamic content that is frequently accessed.
- **Databases**: To cache query results, reducing the need for repeated database access.
- **APIs**: To cache responses from external APIs, reducing the latency for frequently requested data.
- **Operating Systems**: OS-level file system caching to speed up file access.
- **Content Delivery Networks (CDNs)**: CDNs cache content (like videos, images, web pages) at locations closer to end users to improve delivery speed.

7. Cache Consistency & Invalidations:

Maintaining consistency between the cache and the source of truth (e.g., a database) can be challenging. When data changes (e.g., a user updates a profile or a product’s price), you need to make sure that the cache reflects those changes.

Common strategies include:

- **Cache Invalidation**: Explicitly removing or updating the cache when the underlying data changes.
- **Cache Expiry**: Setting a time-to-live (TTL) for cache entries, after which they are automatically invalidated.
- **Write-through/Write-back Caching**: Writing updates to both the cache and the backend system at the same time (write-through) or writing to the cache first and later syncing with the backend (write-back).

8. Cache Replication:

In distributed systems, caching can be spread across multiple servers or nodes to increase scalability and reliability. Each cache node holds a copy of the cached data, and replicas are kept in sync with each other.

- **How replication works**: When a new data entry is cached on one node, the update is propagated to other replicas. This can be done through direct synchronization or using a publish-subscribe mechanism.

9. When Not to Use Caching:

Caching isn't always the best solution. Here are cases where caching might not be ideal:

- **Highly Dynamic Data**: If the data changes frequently (e.g., real-time stock prices or live sports scores), caching might introduce stale data or complexity in maintaining cache consistency.
- **Large Volumes of Data**: If the dataset is huge and accessed infrequently, caching might not be cost-effective, as it could require significant memory resources.
- **Security or Privacy Concerns**: If the data is sensitive or user-specific (e.g., passwords, payment details), caching might introduce security risks unless properly managed.

10. Cache Eviction Policies:

1. **First In, First Out (FIFO)**: The cache evicts the oldest block (the first one accessed) first, regardless of how often or how many times it was accessed.
2. **Last In, First Out (LIFO)**: The cache evicts the most recently accessed block first, regardless of its access frequency.
3. **Least Recently Used (LRU)**: The cache discards the least recently used items first, prioritizing the removal of items that haven't been accessed for the longest time.
4. **Most Recently Used (MRU)**: Contrary to LRU, MRU evicts the most recently accessed items first.
5. **Least Frequently Used (LFU)**: This policy tracks how often each item is accessed, and the items that are accessed the least are evicted first.
6. **Random Replacement (RR)**: The cache randomly selects an item to discard when space is needed, without any regard to the item's usage patterns.

Content Distribution Network (CDN):

CDNs are specialized caches used for sites that serve large amounts of static media. In a typical CDN setup, when a request is made for a piece of static media, the CDN first checks if it has the content cached locally. If the content is not available, the CDN queries the backend servers, caches the file locally, and then serves it to the requesting user.

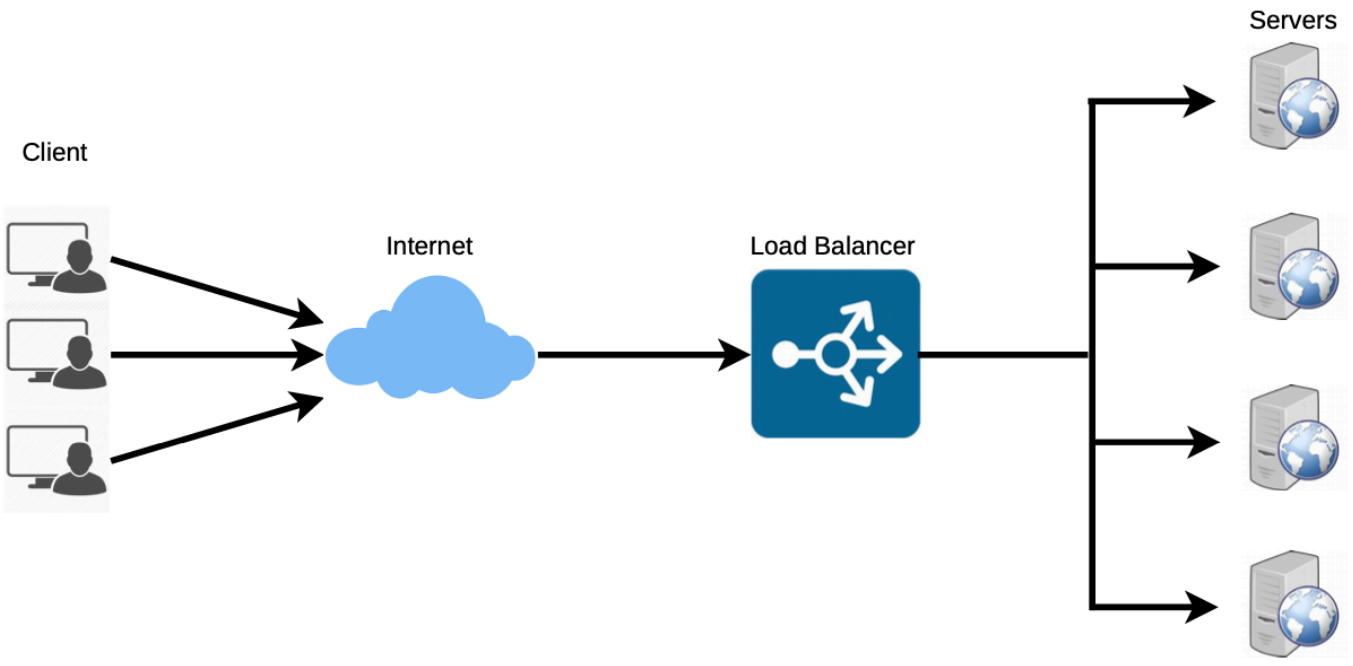
For systems that are not yet large enough to justify their own CDN, a future transition can be eased by serving static media from a separate subdomain (e.g., static.yourservice.com) using a lightweight HTTP server like **Nginx**. Later, the DNS can be switched from your servers to a CDN when scaling becomes necessary.

Load Balancer

A **load balancer** is a critical component in distributed systems and high-availability architectures. It acts as an intermediary between clients and backend servers). Its primary role is to **distribute incoming network traffic across multiple servers** to ensure that no single server becomes overwhelmed with requests. By evenly distributing the load, load balancers improve the performance, reliability, and scalability of a system.

To utilize full scalability and redundancy, we can try to balance the load at each layer of the system. We can add LBs at three places:

• Between the user and the web server
• Between web servers and an internal platform layer, like application servers or cache servers
• Between internal platform layer and database.



Why Do We Use Load Balancers?

1. **Scalability**

2. **High Availability**: Load balancers improve the availability of a system by detecting server failures and rerouting traffic to healthy servers.
3. **Fault Tolerance**: If one server goes down, the load balancer can automatically reroute traffic to other operational servers.
4. **Optimized Resource Utilization**: Load balancing helps ensure that resources on each server are utilized efficiently, avoiding overloading any one server.

**Load Balancing Methods (Algorithms)

Health Checks: Load balancers should only forward traffic to "healthy" backend servers. To monitor the health of a backend server, **health checks** are performed regularly, attempting to connect to the server to ensure it is listening and responsive. If a server fails a health check, it is automatically removed from the server pool, and traffic will no longer be forwarded to it until it passes the health checks and becomes healthy again.

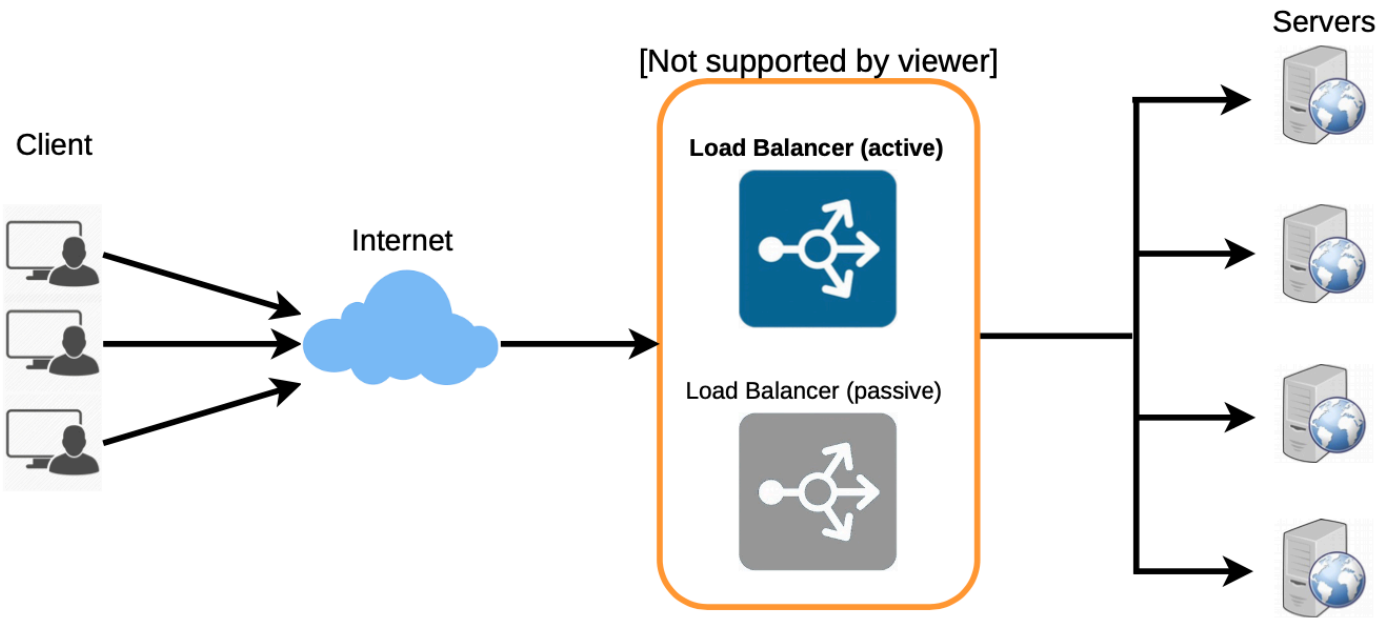
1. **Round Robin**:
   - **How it works**: This is the simplest and most commonly used load balancing method. Requests are distributed equally among all available backend servers in a circular order.
   - **Pros**: Easy to implement, works well when all servers have similar capacity and workload.
   - **Cons**: Doesn’t account for the load or performance of the individual servers. If one server becomes overloaded or slow, the load balancer continues to send requests to it.
2. **Least Connections**:
   - **How it works**: This method sends requests to the server with the fewest active connections. It’s useful when server load can be approximated by the number of concurrent connections.
   - **Pros**: Helps ensure that servers are used more evenly based on their current load.
   - **Cons**: Can be less efficient if the connection duration is very variable (e.g., some requests may take much longer than others).

Load Balancing Between Different Layers:

1. **Between Clients and Application Servers**:
   - A load balancer sits between client requests (e.g., web browsers or mobile apps) and a pool of **application servers** (e.g., web servers, API servers). This ensures that client traffic is distributed evenly across multiple application servers to prevent any one server from becoming a bottleneck.
2. **Between Application Servers and Database Servers**:
   - If the application servers need to query a backend database, a load balancer can be used to distribute database queries across multiple database replicas or instances. This helps in scaling the database layer, especially for read-heavy applications where replication can be used to serve different read requests across multiple database nodes.
3. **Between Application Servers and Cache Servers**:
   - In systems using caching (e.g., Memcached, Redis), a load balancer can distribute cache requests across multiple cache servers. This is useful in systems where the cache layer is distributed or partitioned (e.g., in sharded caching architectures).

Redundant Load Balancers:

The load balancer can be a single point of failure in a system. To mitigate this risk, a second load balancer can be connected to the first, forming a cluster. Both load balancers monitor each other's health and are capable of handling traffic. If the primary load balancer fails, the secondary load balancer automatically takes over, ensuring continuous traffic distribution and failure detection.



Purging

Purging refers to the process of **removing outdated, unnecessary, or unwanted data** from a system or cache. It is typically done to free up storage space, improve performance, and ensure that only relevant or up-to-date information remains available. Purging can apply to a variety of contexts, such as caches, databases, logs, or file systems.

Types of Purging Strategies:

1. **Automatic Purging**: This is when data is automatically removed based on defined policies. For example:
   - Cache items expire after a certain TTL (Time to Live).
   - Old logs are automatically deleted after a specific time frame (e.g., logs older than 30 days).
   - Databases or storage systems automatically clean up old records or files based on age or last access time.
2. **Manual Purging**: In this case, data is manually deleted by an administrator or user. This might be done when:
   - A user wants to free up space.
   - A database administrator wants to clean up records that are no longer needed.
   - A website owner needs to purge outdated content from a CDN or cache.
3. **Conditional Purging**: Purging is done based on specific conditions or events. For example:

- Data is purged after a certain event, such as the expiration of a user session or after an order is marked as completed.
- Data is purged based on system performance or load (e.g., when disk usage exceeds a threshold, older files or logs are purged).

Security and Permissions

Security and **permissions** are foundational concepts that ensure the integrity, confidentiality, and availability of data, as well as proper access control. These concepts help protect systems from unauthorized access, data breaches, and malicious actions.

Authentication and Authorization
Biometrics
Passwords
Multi-Factor Authentication
OAuth

SQL vs NoSQL

When it comes to database technology, there's no one-size-fits-all solution. That's why many businesses rely on both relational and non-relational databases for different needs.

Why SQL

- **ACID Compliance**: SQL databases ensure ACID compliance, which guarantees transaction integrity and consistency, making them ideal for applications like e-commerce or financial systems where data accuracy is critical.
- **Structured, Stable Data**: SQL databases are well-suited for applications with structured, consistent data that doesn't require frequent changes or high scalability demands.

Why NoSQL

1. **Handling Unstructured Data**: NoSQL databases excel at storing large volumes of unstructured or semi-structured data. They don't require predefined schemas, making them flexible and adaptable as data types evolve.
2. **Cloud Scalability**: NoSQL databases like Cassandra are designed for easy scalability across multiple servers and data centers, making them ideal for cloud environments. They enable cost-effective, distributed storage that can seamlessly grow as needed.
3. **Rapid Development**: NoSQL is perfect for fast-paced development cycles. It allows for quick iterations and changes to the data structure without the need for downtime or complex schema updates, unlike relational databases.

2 NoSQL

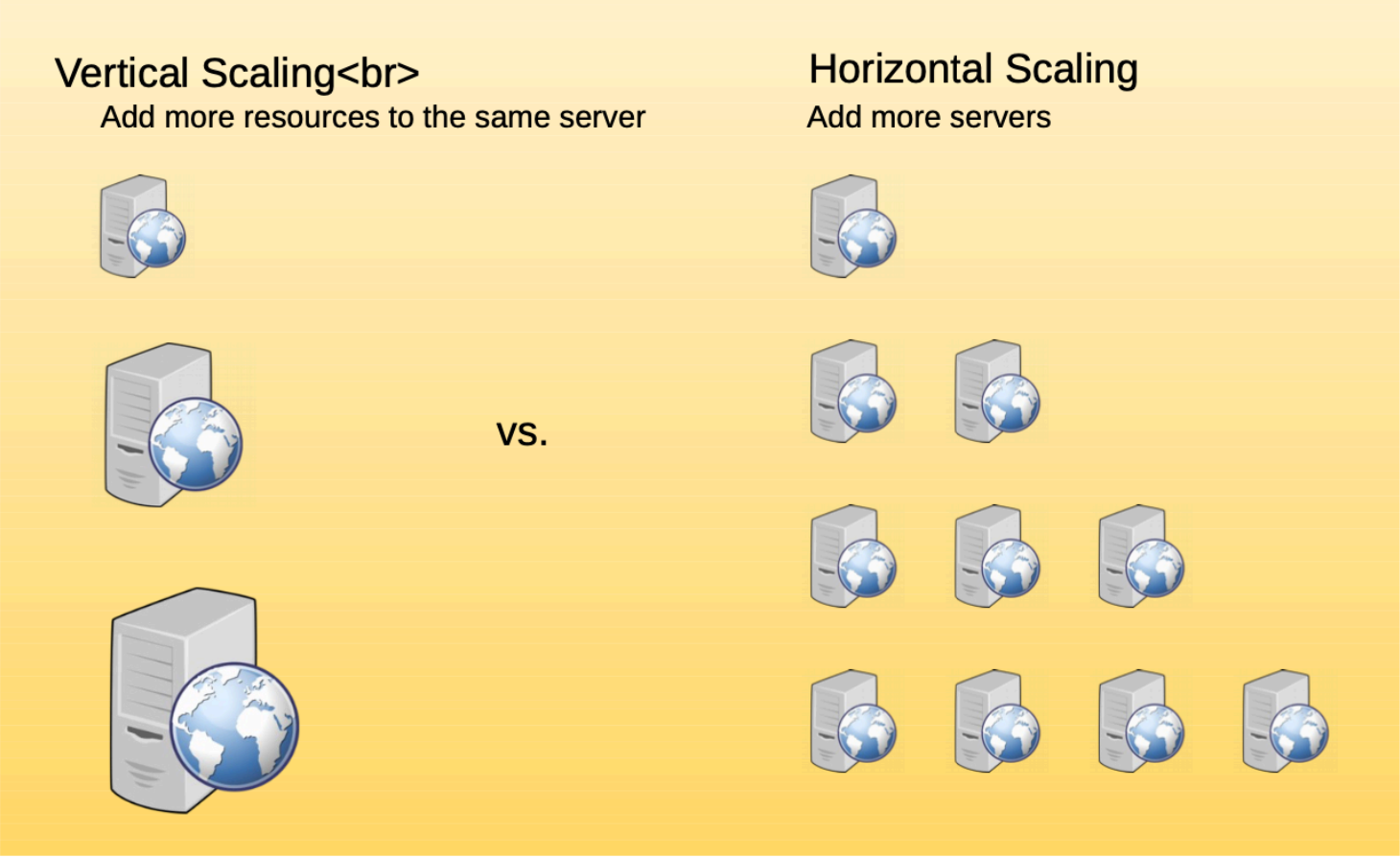
Graph Databases: These databases are designed to store data whose relationships are best represented in a graph structure. Data is organized using **nodes** (entities), **properties** (information about the entities), and **edges** (connections between the entities). Graph databases are ideal for applications where relationships between data points are crucial, such as social networks or recommendation engines. Examples of graph databases include **Neo4j** and **InfiniteGraph**.

Key-Value Stores: Data is stored in an array of key-value pairs. The 'key' is an attribute name which is linked to a 'value'.

Scalability

Scalability refers to the ability of a system, network, or process to handle a growing amount of work, or its potential to accommodate growth. A scalable system can manage increased demand without a significant drop in performance or requiring a complete redesign.

1. **Vertical Scaling (Scaling Up)**:
   - **Definition**: Vertical scaling involves adding more power (CPU, RAM, storage) to an existing server to handle more load. This is often done by upgrading the hardware of a single machine.
2. **Horizontal Scaling (Scaling Out)**:
   - **Definition**: Horizontal scaling involves adding more servers to the system, distributing the load across multiple machines. This is often done in distributed systems, where the system architecture is designed to work across multiple machines



Reliability

Reliability refers to the probability that a system will continue functioning without failure over a specified period. In distributed systems, reliability ensures continuous service delivery even if some hardware or software components fail. This is typically achieved through redundancy, where backup systems or data replicas can take over in case of failures. However, implementing redundancy incurs additional costs. A reliable system prioritizes minimizing downtime and ensuring task completion despite component failures.

Availability

Availability is the percentage of time a system is operational and able to perform its required functions within a specific period. It measures how much time a system, service, or machine remains up and running under normal conditions, factoring in maintainability, repair time, and spares availability.

Reliability vs. Availability:

- **Reliability contributes to availability**: A reliable system is typically available, as it can function without frequent failures.
- **High availability doesn't guarantee reliability**: A system can achieve high availability by minimizing downtime (e.g., quick repairs or having spares), but that doesn't mean it's free from vulnerabilities or failures over time.

Efficiency

Efficiency in a distributed system can be measured through two primary metrics:

1. **Response Time (Latency)**: The time it takes to obtain the first item in a set of results after initiating an operation. It reflects how quickly the system responds to a request.
2. **Throughput (Bandwidth)**: The number of items delivered or processed within a specific time frame (e.g., per second). It measures the system's capacity to handle requests or deliver data.

Key Cost Units:

- **Number of messages**: The total number of messages sent between nodes in the system, irrespective of their size.
- **Size of messages**: The volume of data exchanged between nodes.

Serviceability or Manageability

Serviceability or **Manageability** is another important consideration when designing a distributed system. It refers to the simplicity and speed with which a system can be repaired or maintained. If the time to fix a failed system increases, then availability will decrease.

Things to consider for manageability include the ease of diagnosing and understanding problems when they occur, the ease of making updates or modifications, and how simple the system is to operate (i.e., does it routinely operate without failure or exceptions?).

Early detection of faults can decrease or avoid system downtime. For example, some enterprise systems can automatically call a service center (without human intervention) when the system experiences a system fault.

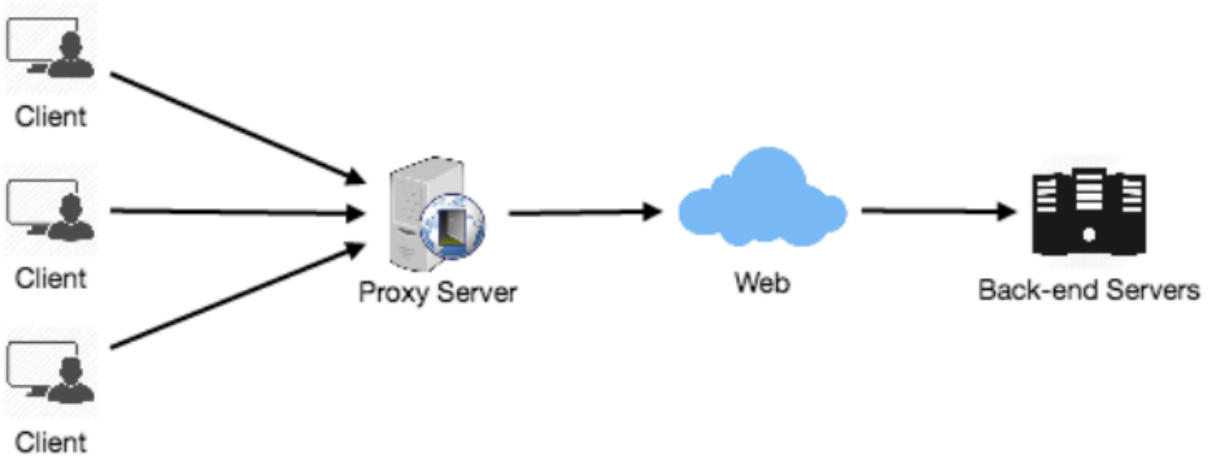
Proxy Server

A **proxy server** acts as an intermediary between the client and the back-end server. When a client requests a service (like a web page, file, or connection), the request goes through the proxy server, which forwards the request to the appropriate server and returns the response to the client. Proxy servers are typically used to filter requests, log activities, or transform requests (e.g., by adding/removing headers, encrypting, decrypting, or compressing resources).

One key benefit of a proxy server is its ability to **cache** content. When multiple clients request the same resource, the proxy can cache it and serve it directly to clients, reducing the load on the back-end server.

Types of Proxy Servers

Proxies can be placed on the client's local server or anywhere between the client and the remote servers. Below are the main types of proxy servers:



1. Open Proxy

An **open proxy** is accessible by any Internet user, allowing anyone to use it to forward requests to other servers. Open proxies are typically less secure and can be used for malicious purposes, but they can also offer anonymity. There are two common types of open proxies:

- **Anonymous Proxy**:
  - **Description**: This proxy hides the user's IP address but reveals its own identity as a server. While it is not entirely secure and can be easily detected, it is useful for users who wish to hide their IP address.
- **Transparent Proxy**:
  - **Description**: This proxy identifies itself and also reveals the original IP address of the client through HTTP headers. Its main benefit is caching frequently accessed websites, which can speed up access and reduce bandwidth usage.

2. Reverse Proxy

A **reverse proxy** works on behalf of one or more back-end servers. It receives requests from clients and retrieves resources from the server(s), then returns the resources to the client as if they originated from the reverse proxy itself. This type of proxy is often used for load balancing, caching, security, or to enable access to multiple servers through a single IP address.

Redundancy and Replication

Redundancy

- **Definition**: Redundancy refers to the duplication of critical system components or functions to increase **reliability** and ensure **availability**. It is typically used as a **backup** or **fail-safe** mechanism to improve performance and protect against system failures.
- **Example**: If a file is stored on a single server and that server fails, the file is lost. To prevent this, redundant copies of the file can be stored on different servers, so even if one server fails, the file remains accessible.
- **Key Role**: Redundancy helps to **remove single points of failure** in a system, ensuring **continuity**. For instance, if two instances of a service are running in production and one fails, the system can **failover** to the other instance, maintaining service availability.

Replication

- **Definition**: Replication is the process of copying data or resources between systems (servers, databases, etc.) to maintain **consistency** and ensure **fault-tolerance** and **accessibility**. It helps to keep redundant components synchronized.
- **Application in Databases**: In **database management systems (DBMS)**, replication typically follows a **master-slave** structure:
  - The **master** database handles updates and changes.
  - These changes are **replicated** to **slave** databases, ensuring that all copies of the data are up-to-date.
  - Each slave acknowledges the receipt of updates, which allows subsequent updates to be sent to other replicas.
- **Benefits**: Replication improves **data availability**, ensures **data consistency**, and enhances **fault tolerance** by distributing copies across multiple systems, ensuring that even if one database fails, others are available.

Web Sockets

WebSocket provides full-duplex communication channels over a single TCP connection. It establishes a persistent connection between a client and a server, allowing both parties to send data at any time. The client initiates the WebSocket connection through a process known as the WebSocket handshake. If this handshake succeeds, the server and client can exchange data in both directions at any time.

The WebSocket protocol enables communication between a client and a server with lower overhead, facilitating real-time data transfer between the two. This is achieved by providing a standardized way for the server to send content to the browser without the client needing to request it, while allowing for continuous messages to be passed back and forth, keeping the connection open. In this way, a two-way (bi-directional) ongoing conversation can take place between the client and server.

WebSockets Protocol

Server-Sent Events (SSEs)

In **Server-Sent Events (SSEs)**, the client establishes a persistent, long-term connection with the server. The server then uses this connection to send data to the client. If the client wants to send data back to the server, it requires a different technology or protocol to do so.

SSE Flow:

1. The client requests data from the server using a regular HTTP request.
2. The requested webpage opens a connection to the server.
3. The server sends data to the client whenever new information is available.

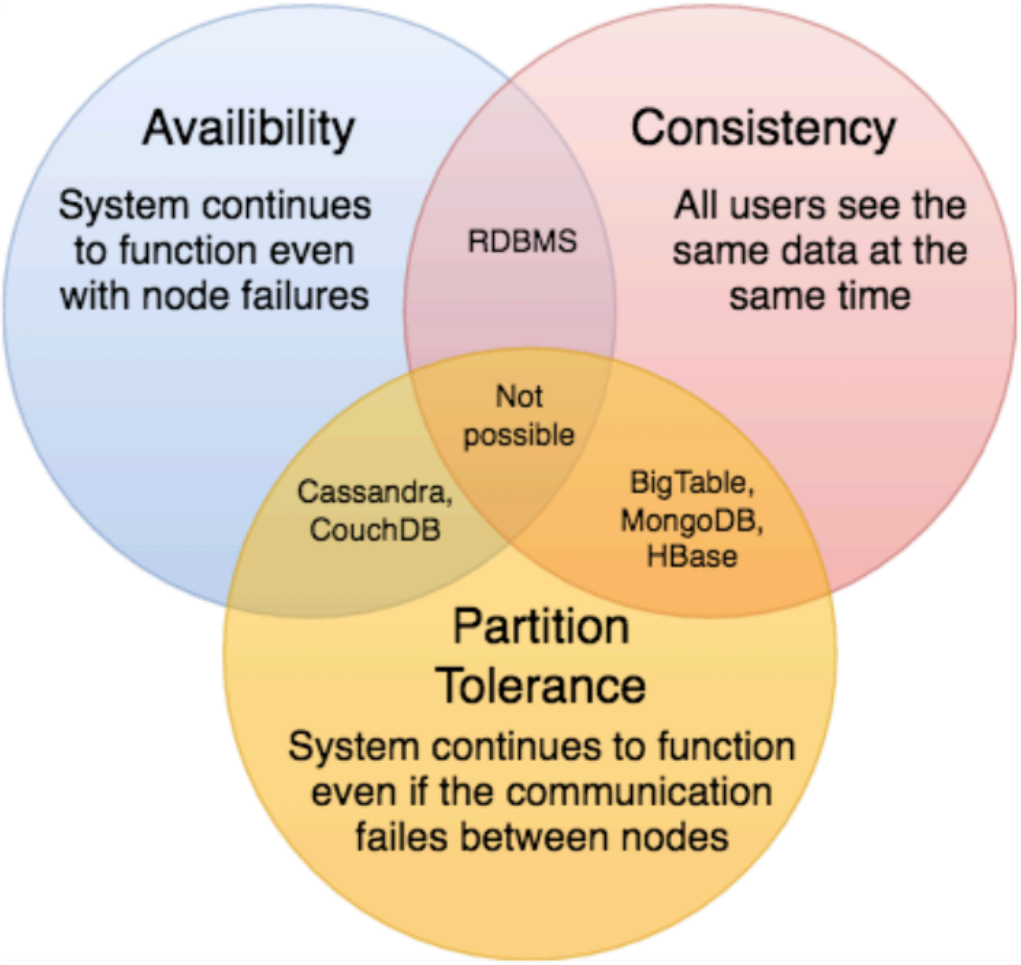
CAP Theorem

CAP theorem states that it is impossible for a distributed software system to simultaneously provide more than two out of three of the following guarantees (CAP): Consistency, Availability, and Partition tolerance.
When we design a distributed system, trading o@ among CAP is almost the first thing we want to consider. CAP theorem says while designing a distributed system we can pick only two of the following three options:

Consistency: All nodes see the same data at the same time. Consistency is achieved by updating several nodes before allowing further reads.

Availability: Every request gets a response on success/failure. Availability is achieved by replicating the data across different servers.

Partition tolerance: The system continues to work despite message loss or partial failure. A system that is partition-tolerant can sustain any amount of network failure that doesn’t result in a failure of the entire network.


Availibility
System continues to function even with node failures
Consistency
All users see the same data at the same time
RDBMS
Not possible
Cassandra, CouchDB
BigTable, MongoDB, HBase
Partition Tolerance
System continues to function even if the communication failes between nodes