Performance vs. Scalability

✔ Performance Problem: The system is slow for a single user.

✔ Scalability Problem: The system is fast for one user but slow under heavy load.

✔ Scalable System:

Performance increases proportionally as more resources are added.

Can handle more requests or larger datasets as needed.

Latency vs. Throughput

✔ Latency: Time taken to complete a single operation (e.g., response time).

✔ Throughput: Number of operations completed per unit time.

✔ Goal: Maximize throughput while keeping latency acceptable.

Example: Communication Device Computation

✔ Clock Frequency: 100MHz

✔ Time for Computation: 1000ns

✔ Throughput: 640 Mbits/sec

✔ Word Width: 64 bits

Latency = Time Available * (Clock cycles per second / 1 second)

= 1000 ns * (100 * 10^6 clock cycles / 10^9 ns)

= 100 clock cycles

🔹 It takes 100 clock cycles for one computation.

Throughput = (Data Rate (bits per s) / word size (bits)) * (1s / Clock cycles per s)

= (640 * 10^6 bits/s / 64 bits) * (1s / 100 * 10^6 clock cycles)

= 0.1 words per clock cycle

🔹 The system processes one word every 10 clock cycles.

Consistency vs. Availability

🔹 Consistency (Accuracy of Data in Reads & Writes)

✔ Definition: Ensures that every read reflects the latest write or returns an error.

Type	Description	Examples
Weak Consistency	Reads may or may not reflect the latest writing.	UDP, VoIP, Video Chat
Eventual Consistency	Reads will eventually reflect the latest write, but not immediately.	DNS, Email
Strong Consistency	Reads always reflect the latest writing.	File Systems, Databases with Transactions

🔹 Availability

✔ Definition: Guarantees every request gets a response, even if it's not the most recent data.

✔ Key Techniques:

Failover & Replication to improve availability.

✔ Failover Types:

Type	Description
Active-Passive	Standby server (passive) monitors the active server. If failure occurs, the passive takes over.
Active-Active	Both servers share traffic, balancing the load dynamically.

📌 High availability prioritizes uptime, while consistency ensures accurate data.

📌 Key Points

✔ Consistency vs. Availability is a trade-off (inversely related).

✔ Strong consistency → Ensures data accuracy (e.g., financial systems, file storage).

✔ High availability → Keeps the service running without downtime (e.g., DNS, chat services).

Replication

1️⃣ Master-Slave Replication

✔ How it works:

Master handles reads & writes and replicates writes to slave nodes.

Slaves handle read-only requests (improves performance).

Slaves can replicate further in a tree-like structure.

✔ Failure Handling:

If the master fails, the system operates in read-only mode until a slave is promoted to master.

2️⃣ Master-Master Replication

✔ How it works:

Multiple masters handle reads & writes and coordinate to sync data.

✔ Failure Handling:

If one master fails, the system continues to operate normally with the other master.

3️⃣ Replication Cons (Challenges & Trade-offs)

✔ Data Loss Risk:

If the master fails before replicating, recent writes may be lost.

✔ Replication Overhead:

Writes must be replayed to replicas, causing delays when there are too many writes.

✔ Replication Lag:

More read replicas increase replication lag (delays in syncing data).

✔ Complexity & Cost:

Requires more hardware and more complex coordination.

📌 Replication improves performance & availability, but comes with data consistency challenges.

Partition Tolerance (P)

✔ Definition: The system continues operating even when network failures cause partitioning.

✔ Reality: Networks are unreliable, so partition tolerance is a must.

✔ Trade-off: Must choose between consistency (C) and availability (A) → CAP theorem.

2️⃣ **CAP Theorem: Choose Two**

Type	**Guarantees**	**Trade-off**	**Example**
CP (consistency + Partition Tolerance)	Data is always consistent across nodes	Some data may be unavailable if a partition occurs	MongoDB
AP (availability + Partition Tolerance)	System always responds, even with network failures.	Some responses may be outdated (not the latest data)	DNS

✔ MongoDB (CP - Consistency + Partition Tolerance)

1. Ensures strong consistency → Every read reflects the most recent write.
2. If a network partition occurs, some nodes reject requests to maintain data accuracy.
3. Trade-off: Availability is sacrificed during network failures.

Use case: Suitable for systems requiring strict data consistency (e.g., databases, financial systems).

✔ DNS (AP - Availability + Partition Tolerance)

1. Ensures high availability → Always responds, even during network failures.
2. If a network partition occurs, some users may receive outdated data due to caching.

3. Trade-off: Consistency is sacrificed, but service remains accessible.

Use case: Suitable for systems where availability is more important than strict consistency (e.g., DNS, web services).

📌MongoDB prioritizes data accuracy (CP), DNS prioritizes continuous service availability (AP).

Vertical Scaling vs. Horizontal Scaling

1️⃣Vertical Scaling (Scaling Up)

✔ Definition: Upgrading a single machine by adding more CPU, RAM, or storage to handle higher loads.

✔ Examples:

Databases: Upgrading a MySQL/PostgreSQL server for better performance.

Web Servers: Increasing CPU/RAM on an Apache or Nginx server.

✔ Pros:

Simpler architecture (no need for distributed systems).

Easier to manage and maintain compared to multiple servers.

✔ Cons:

Hardware limits (a single machine has a maximum capacity).

Downtime risk (upgrading may require taking the system offline).

Expensive (high-end hardware costs more).

2️⃣Horizontal Scaling (Scaling Out)

✔ Definition: Adding more machines (servers) to distribute the workload across multiple nodes.

✔ Examples:

Web Apps: Facebook/Twitter use multiple servers behind a load balancer.

Databases: NoSQL databases (Cassandra, MongoDB) distribute data across many nodes.

Cloud Services: Kubernetes, Docker Swarm manage workloads across multiple machines.

✔ Pros:

High availability (if one server fails, others keep running).

Infinite scalability (easily add more machines as needed).

Cost-effective (commodity hardware is cheaper than high-end machines).

✔ Cons:

More complex (requires load balancing, replication, and distributed systems).

Data consistency issues (keeping nodes synchronized is difficult).

📌 Vertical scaling is simple but limited, while horizontal scaling is more complex but allows unlimited growth.

Load Balancing

✔ Load balancing distributes incoming client requests across multiple computing resources, such as application servers and databases, to ensure efficient performance and reliability.

2️⃣Pros of Load Balancing

✔ Prevents requests from going to unhealthy servers, ensuring reliability.

✔ Avoids overloading resources, improving performance.

✔ Eliminates a single point of failure, increasing system availability.

3️⃣Cons of Load Balancing

✔ Can become a performance bottleneck if the load balancer itself lacks resources or is misconfigured.

✔ Adds complexity to system architecture.

✔ A single load balancer can become a failure point, requiring multiple load balancers for redundancy, which further increases complexity.

📌 Load balancing improves availability and performance, but requires proper configuration and redundancy to avoid new failure points.

1️⃣Process Flow:

1. Pick a worker to forward the request using a load balancing strategy:
   a. Random: Assigns requests randomly to workers.
   b. Round Robin: Distributes requests sequentially across workers.
   c. Least Busy: Sends the request to the worker with the fewest active tasks.
   d. Sticky Session / Cookies: Ensures that the same client is always assigned to the same worker.
   e. By Request Parameters: Uses specific request attributes (e.g., user ID, session ID) to determine which worker should process the request.
2. Wait for the worker's response after processing the request.
3. Forward the response to the client once the worker completes processing.

2️⃣Components:

✔ Client → Sends a request to the system.

✔ Dispatcher (Load Balancer) → Distributes incoming requests across available workers.

✔ Worker Pool → A group of worker servers that process requests and interact with databases.

3️⃣Purpose & Benefits:

✔ Ensures efficient request distribution and prevents any single worker from being overloaded.

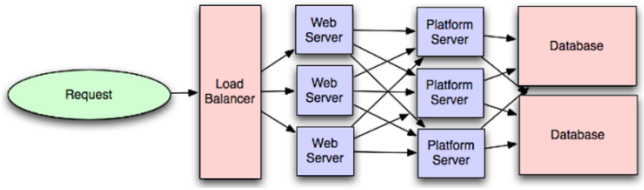
✔ Improves system scalability by allowing more workers to be added dynamically.

✔ Enhances reliability & fault tolerance → If a worker fails, the dispatcher can redirect traffic to other available workers.

📌 This system ensures load balancing, high availability, and better performance by distributing client requests among multiple backend workers efficiently.

Application Layer

Separating out the web layer from the application layer



Database

1️⃣Federation (Function-Based Database Splitting)

✔ Instead of a single, monolithic database, the system is split into multiple databases by function (e.g., users, products, forums).

✔ Benefits: Smaller databases fit more data into memory, improving performance.

✔ Drawbacks: Not effective if the schema requires large, interconnected functions or tables.

2️⃣Sharding (Data-Based Database Splitting)

✔ Data is split across multiple databases (shards), where each database stores only a subset of the total data.
✔ Example: A user table can be sharded by last name initials or geographic location.
✔ Drawbacks: Joining data across shards is complex, making queries harder

3️⃣ SQL vs. NoSQL - When to Use Each?

SQL (Relational Databases - Structured & Consistent Data)

✔ Best for data integrity and strict transactions.
✔ Use cases:
- Banking & Financial Systems (strong consistency & security).
- ERP (Enterprise Resource Planning) software.
- E-commerce, CMS (structured data & relationships).

NoSQL (Non-Relational Databases - Scalable & Flexible Data)

✔ Best for schema-less and dynamic data for faster scalable data
✔ Use cases:
- Social Media Platforms (flexible, unstructured user data).
- Real-Time Applications (chat apps, analytics dashboards).
- IoT & Big Data (handling massive, rapidly changing and irregular data).
- Distributed Systems (microservices, cloud applications).

📌 Federation & sharding help scale databases. SQL is suitable when data consistency is critical, while NoSQL is ideal for scalability and flexibility.

Cache

✔ Caching improves page load times and reduces the load on servers and databases by storing frequently accessed data in a fast-access memory layer.
✔ Types of Caching:
- Client Caching → Stores data on the user's device (e.g., browser cache).
- CDN Caching → Caches static content on edge servers to speed up delivery.
- Web Server Caching → Stores responses at the server level (e.g., Nginx, Apache).
- Database Caching → Caches database query results (e.g., Redis, Memcached).
- Application Caching → Stores frequently used data in memory within an application.

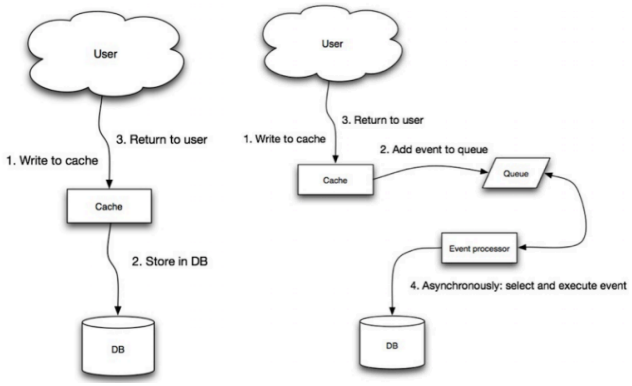
2️⃣ When to Update Cache?

✔ Cache-aside (Lazy Loading)
1. Look for the entry in the cache → If cache miss, proceed to the next step.
2. Load the entry from the database.
3. Add the entry to the cache.
4. Return the entry to the user.

✔ Write-through Caching
Data is written to both the cache and the database at the same time.
Ensures data consistency but may introduce latency due to synchronous writes.

✔ Write-behind Caching (Illustrated in the Image)

Data is written to the cache first.
The write operation is then added to a queue for asynchronous processing.
A separate event processor handles updating the database in the background.
Faster writes for the user but may lead to data loss if the system crashes before writing to the database.



4️⃣ Key Takeaways

✔ Cache reduces database load and speeds up response times.
✔ Cache-aside ensures efficient memory usage but may lead to initial cache misses.
✔ Write-through caching keeps data consistent but slows down write operations.
✔ Write-behind caching improves write performance but introduces the risk of data loss if the queue fails before processing.