LLD & HLD TOPICS

LLD TOPICS:

Design Patterns - YouTube

1. S.O.L.I.D Principles
2. Strategy Pattern
3. Observer Pattern
4. Design Notify-Me Button Functionality
5. Decorator Pattern
6. Design Pizza Billing System
7. Factory Pattern
8. Design Parking Lot
9. Abstract Factory Pattern
10. Design Snake n Ladder game
11. Chain of Responsibility Pattern
12. Design Elevator System
13. Proxy Pattern
14. Design Car Rental System
15. Null Object Pattern
16. Design Logging System
17. State Pattern
18. Design Tic-Tac-Toe game
19. Composite Pattern
20. Design BookMyShow & Concurrency handling
21. Adapter Pattern
22. Design Vending Machine
23. Singleton Pattern
24. Design ATM
25. Builder Pattern
26. Design Chess game
27. Prototype Pattern
28. Design File System
29. Bridge Pattern
30. Design Splitwise
31. Façade Pattern
32. Splitwise Simplify Algorithm / Optimal Accounting Balancing
33. Flyweight Pattern
34. Design CricBuzz / CricketInfo
35. Command Pattern
36. Design True Caller
37. Interpreter Pattern
38. Design Car Booking Service like Ola, Uber
39. Iterator Pattern
40. Design Online Hotel Booking System
41. Mediator Pattern
42. Design Library Management System
43. Memento Pattern
44. Design Traffic Light System
45. Template Method Pattern
46. Design Meeting Scheduler
47. Visitor Pattern
48. Design Online Voting System
49. Design Inventory Management System
50. Design Cache Mechanism
51. Design LinkedIn
52. Design Amazon
53. Design Airline Management System
54. Design Stock Exchange System
55. Design Learning Management System
56. Design a Calendar Application
57. Design (LLD) Payment System
58. Design (LLD) Chat based system
59. Design Food delivery app like Swiggy and Zomato
60. Design Community Discussion Platform
61. Design Restaurant Management System
62. Design Bowling Alley Machine
63. Design (LLD) Rate Limiter

HLD TOPICS:

1. Learn About Network Protocols (TCP, Websocket, HTTP etc.)
2. Client-Server Vs Peer 2 Peer Architecture
3. C.A.P Theorem
4. Microservices Imp. Design Patterns (SAGA pattern, Strangler Pattern)
5. Scale from 0 to Million Users
6. Design Consistent Hashing
7. Design URL Shortening
8. Back of the Envelope Estimation
9. Design Key-Value Store
10. SQL vs NoSQL, When to Use Which DB
11. Design WhatsApp
12. Design Rate Limiter
13. Design Search Autocomplete System / Typeahead System
14. Understand Message Queue , Kafka etc.
15. What is Proxy Servers
16. What is CDN
17. Storage types:
18. (Block Storage, File Storage, Object Storage (S3) , RAID)
19. File System
20. (Google File System, HDFS)
21. Bloom Filter
22. Merkle Tree , Gossiping Protocol
23. Caching
24. (Cache Invalidation, Cache eviction)
25. How to Scale Database

26. - Sharding (Horizontal and Vertical)
27. - Partitioning
28. - Replication, Mirroring
29. - Leader Election
30. - Indexing etc.
31. Design Notification System
32. Design Pastebin
33. Design Twitter
34. Design Dropbox
35. Design Instagram
36. Design YouTube
37. Design Google Drive
38. Design Web Crawler
39. Design Facebook News Feed / News Feed System
40. Design Ticket Master
41. Design NearByFriends or Yelp

System Design Roadmap

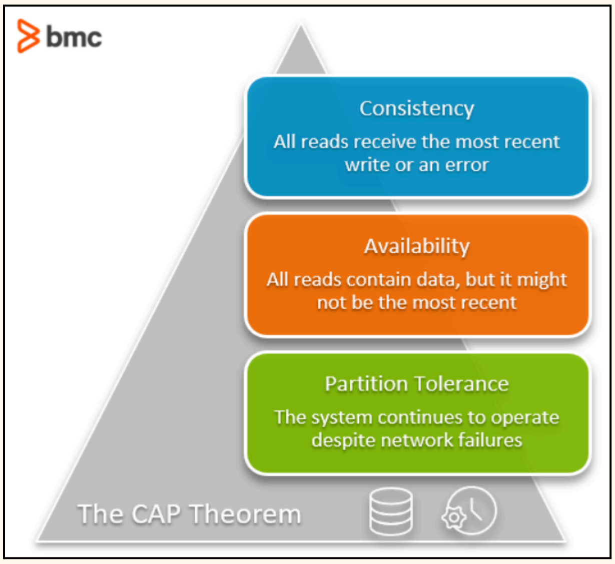
LATENCY VS THROUGHPUT:

Aspect	Throughput	Latency
Definition	The number of tasks completed in a given time period.	The time it takes for a single task to be completed.
What does it measure?	Throughput measures the volume of data that passes through a network in a given period. Throughput impacts how much data you can transmit in a period of time.	Latency measures the time delay when sending data. A higher latency causes a network delay.
How to measure?	Manually calculate throughput by sending a file or using network testing tools.	Calculate latency by using ping times.
Unit of measurement	Typically measured in operations per second or transactions per second. Megabytes per second (MBps).	Measured in time units such as Milliseconds (ms) or seconds(s).
Impacting factors	Bandwidth, network processing power, packet loss, and network topology.	Geographical distances, network congestion, transport protocol, and network infrastructure.
Relationship	Inversely related to latency. Higher throughput often corresponds to lower latency.	Inversely related to throughput. Lower latency often corresponds to higher throughput.
Impact on System	Reflects the overall system capacity and ability to handle multiple tasks simultaneously.	Reflects the responsiveness and perceived speed of the system from the user's perspective.
Example	A network with high throughput can transfer large amounts of data quickly.	Low latency in gaming means minimal delay between user input and on-screen action.

CAP THEOREM:

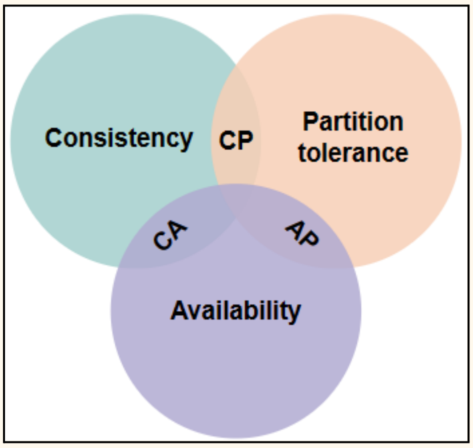
The CAP theorem states that a distributed system can only provide two of three properties simultaneously: consistency, availability, and partition tolerance.

The CAP Theorem is comprised of three components (hence its name) as they relate to distributed data stores:


bmc
Consistency
All reads receive the most recent write or an error
Availability
All reads contain data, but it might not be the most recent
Partition Tolerance
The system continues to operate despite network failures
The CAP Theorem

1. **Consistency:** This refers to the requirement that all nodes in a distributed system see the same data at the same time. In other words, if a write operation is successful, all subsequent read operations should return the updated value.
2. **Availability:** This refers to the requirement that a distributed system must always be available to respond to client requests, even in the event of a node failure or network partition.
3. **Partition tolerance:** This refers to the requirement that a distributed system must continue to function even when network partitions occur, meaning that nodes in different parts of the network are unable to communicate with each other.

CAP Theorem NoSQL Databases:

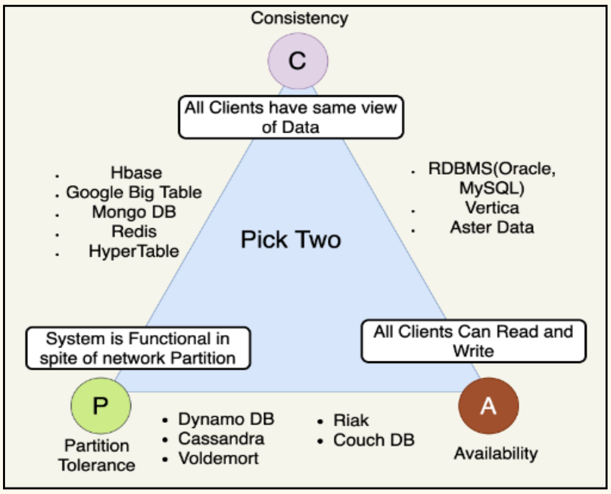


CA Databases:

CA databases enable consistency and availability across all nodes. Unfortunately, CA databases can't deliver fault tolerance. In any distributed system, partitions are bound to happen, which means this type of database isn't a very practical choice. That being said, you still can find a CA database if you need one. Some relational databases, such as PostgreSQL, allow for consistency and availability. You can deploy them to nodes using replication.

CP Databases:

CP databases enable consistency and partition tolerance, but not availability. When a partition occurs, the system has to turn off inconsistent nodes until the partition can be fixed. MongoDB is an example of a CP database. It's a NoSQL database management system (DBMS) that uses documents for data storage. It's considered schema-less, which means that it doesn't require a defined database schema. It's commonly used in big data and applications running in different locations. The CP system is structured so that there's only one primary node that receives all of the write requests in a given replica set. Secondary nodes replicate the data in the primary nodes, so if the primary node fails, a secondary node can stand-in.



AP Databases:

AP databases enable availability and partition tolerance, but not consistency. In the event of a partition, all nodes are available, but they're not all updated. For example, if a user tries to access data from a bad node, they won't receive the most up-to-date version of the data. When the partition is eventually resolved, most AP databases will sync the nodes to ensure consistency across them. Apache Cassandra is an example of an AP database. It's a NoSQL database with no primary node, meaning that all of the nodes remain available. Cassandra allows for eventual consistency because users can resync their data right after a partition is resolved.

ACID TRANSACTION:

ACID transactions refer to a set of properties that are designed to ensure the reliability and consistency of database transactions. The term "ACID" stands for **Atomicity, Consistency, Isolation, and Durability**, which are the four key properties of an ACID transaction. Essentially, ACID transactions guarantee that database operations are executed correctly, and if there is any kind of failure, the database can recover to a previous state without losing any data or impacting the consistency of the data. In other words, ACID transactions provide a high level of assurance that database transactions will be processed reliably and that data will be stored accurately and consistently.

- **Atomicity:**Atomicity in ACID transactions guarantees that a transaction is treated as a single, indivisible unit of work. If any part of the transaction fails, the entire transaction must be rolled back. This ensures that the database remains in a consistent state, regardless of any failures that may occur during the transaction.
  Examples: If a transaction fails after completing one part but before another (e.g., transferring money from account X to Y), it could lead to an inconsistent database state. To maintain correctness, transactions must execute entirely.
- **Consistency:** Consistency ensures that the database remains in a valid state before and after the transaction. In other words, the database schema must satisfy all constraints and rules, and any transaction that violates these constraints must be rolled back to maintain the consistency of the database. This ensures that the database maintains its integrity and the data remains accurate and reliable.
  Example: If the total amount before and after a transaction remains consistent (e.g., total before = 700, total after = 700), the database is consistent.
- **Isolation:** This property ensures that each transaction operates independently of other transactions, which means that a transaction's effects should only become visible to other transactions after it has been committed. This property prevents interference and conflicts between concurrent transactions, and helps maintain the integrity and consistency of the database. It's important to note that different levels of isolation can be configured for transactions, depending on the specific requirements of the application and the database system being used.
- **Durability:** This characteristic makes sure that, even in a system failure, the changes made to the database during a transaction are irreversible. Any changes made after a transaction is committed must persist, even if the system is destroyed or loses power.

How do acid transactions work?

ACID transactions maintain data integrity by adhering to a set of steps. The steps described are a common way that databases implement ACID transactions, but there could be variations or differences in implementation depending on the specific database system being used.

- **Begin Transaction:** A BEGIN TRANSACTION statement declaration initiates a transaction and establishes a savepoint from which the transaction can be rolled back if necessary.
- **Execute operations:** All operations within the transaction are executed one at a time. The database validates each operation to ensure it complies with the constraints and schema.
- **Commit or Rollback:** Following the successful completion of all operations, the transaction is committed using the COMMIT statement. The transaction is rolled back to the savepoint established at the beginning of the transaction if any operation fails.

Example of an acid transaction in action -

Consider a banking app where a user wishes to transfer funds from one account to another, where the operation's transaction might look like this:
BEGIN TRANSACTION – An example of withdrawing money from the bank using a cheque, pay order, or through an ATM.
Deduct the transfer amount from the source account.
Add the transfer amount to the destination account.
COMMIT – Updating the record of the transaction carried out by the customer.
The transaction is rolled back, and the database is restored to its initial state if any of its operations fail, such as if the source account does not have enough funds.

Acid transactions use cases:

1. **Banking:** Banks use ACID transactions to ensure that payments and other financial transactions are handled accurately and securely. For example, when a customer withdraws money from an ATM, an ACID transaction is executed to update their account balance and record the transaction. The transaction is atomic, meaning it succeeds or fails, and the account balance remains constant.
2. **Healthcare systems:** Healthcare systems use ACID transactions to help guarantee that patient records are updated accurately and that private medical data is protected. Electronic health records (EHRs) contain personal information about patients that must be accurate and consistent. For instance, an ACID transaction occurs when a doctor updates a patient's medication in the EHR to ensure the data is updated atomically, consistently, and durably.
3. **E-commerce applications:** E-commerce applications use ACID transactions to make sure that customer orders are processed correctly, and that inventory levels are updated correctly. For example, an ACID transaction is carried out when a customer purchases an item to update the inventory records and guarantee that the transaction is atomic, consistent, isolated, and durable.

Advantages and disadvantages of using acid transactions:

Advantages	**Disadvantages**
Data integrity– Even if a transaction fails, ACID transactions guarantee that the database will remain in a consistent state. It contributes to data reliability and integrity.	Overhead– The performance of databases is affected by the extra processing overhead required by ACID transactions.
Consistency– The validity of the database is ensured both before and after an ACID transaction. It contributes to database consistency.	Deadlocks– Multiple transactions waiting for each other to release resources can cause deadlocks. Deadlocks can be difficult to resolve and have an impact on database reading and retrieval performance.
Isolation– ACID transactions ensure that each transaction is independent of the others. It also contributes to preserving data integrity by preventing interference between concurrent transactions.	Scalability– ACID transactions can be difficult to implement in large-scale distributed systems that require performance and scalability.

Durability– ACID transactions ensure that database changes made during a transaction are irreversible, even in a system failure. It contributes to data reliability.	Similar data update- When several transactions are running concurrently, they might clash if they attempt to modify the same data at the same time. Thus,one transaction might need to wait for another transaction to finish before it can move forward, which would reduce system performance and increase latency.

Alternatives to acid transactions:

ACID transactions provide several benefits for ensuring data reliability, consistency, isolation, and durability. However, they may not be the best fit for all applications. In such cases, there are various alternative transaction models and theorems available that can be considered instead of ACID. These include:

- **BASE (Basically Available, Soft state, Eventually consistent):**
  BASE isn't necessarily a replacement for ACID transactions, but rather an alternative model for handling distributed systems that cannot guarantee immediate consistency. BASE emphasizes availability and partition tolerance more than consistency. This model trades off near-term consistency for long-term stability. Although it assumes that all data will eventually become consistent, it cannot guarantee this. This approach is appropriate for high-volume distributed systems because it provides greater scalability and availability. It is often used in conjunction with NoSQL databases, which prioritize scalability and availability over strict consistency requirements.
- **CAP (Consistency, Availability, Partition tolerance):**
  The CAP theorem states that in a distributed system, it is impossible to guarantee all three of Consistency, Availability, and Partition tolerance. However, it does not suggest sacrificing consistency for availability and partition tolerance. In fact, the theorem poses a trade-off between consistency and partition tolerance. This means that in the event of a network partition, you must choose between consistency and partition tolerance. The CAP theorem is not truly an alternative transaction model but a theoretical framework for understanding the limitations of distributed systems. Transaction models, such as BASE, can be used in conjunction with the principles of CAP to design and implement distributed systems.
- **NoSQL databases:**
  NoSQL databases do not impose rigid consistency standards and prioritize performance and scalability over immediate consistency. They are often used in applications that require high throughput, and where data consistency is not critical. While relational databases ensure desirable ACID properties, NoSQL databases are more effective at handling large and complex data sets. Additionally, BASE properties can perform even better for a wide range of applications, although ACID is not always guaranteed in this case.

Acid transactions in distributed systems:

Distributed systems comprise numerous computers that interact with each other to deliver a single service. ACID transactions can be difficult to implement in a distributed system because they are made up of multiple nodes that are geographically dispersed and communicate over a network.

Challenges of implementing acid transactions in distributed systems:

Distributed systems consist of multiple computers that collaborate to provide a single service. ACID transactions can be challenging to implement in distributed systems because they comprise multiple nodes that are geographically dispersed and communicate over a network.

- **Network latency:** In distributed systems, network latency can impact the performance of ACID transactions. Longer transaction times and higher overhead can result from network communication delays.
- **Consistency:** Maintaining consistency across all nodes in a distributed system can be challenging. A distributed system's nodes might store different versions of the same data, which could result in discrepancies.
- **Availability:** Keeping a distributed system available can be difficult. Nodes could fail, and it might be challenging to maintain the system's responsiveness.
- **Scalability:** As the number of nodes in a distributed system increases, it becomes more difficult to maintain consistency and availability.

Solutions for maintaining acid properties in distributed systems:

ACID transactions are difficult to maintain in distributed systems due to factors such as network latency, consistency, availability, and scalability. To address these challenges, several solutions have been developed, including:

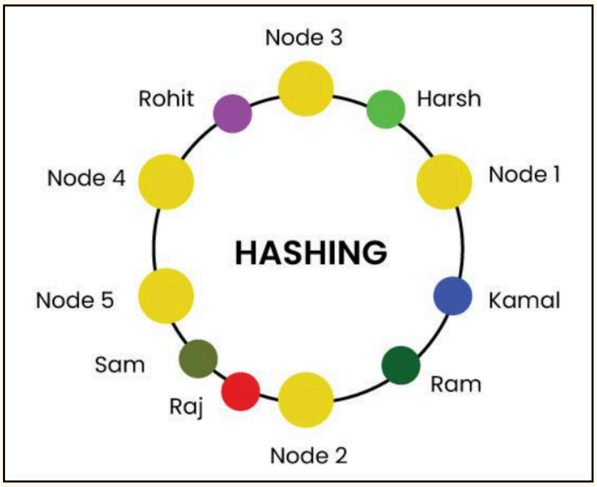
- Two-phase commit: This protocol ensures that all nodes in a distributed system agree to commit a transaction before it is committed, ensuring data consistency and agreement on the transaction's outcome.
- Multi-Version Concurrency Control (MVCC): This technique manages data concurrency in a distributed system by allowing each transaction to access the appropriate data version, enabling multiple versions of the same data to coexist.
- Replication: In a distributed system, replication involves keeping multiple copies of the same data on various nodes, reducing network latency and increasing availability.
- Sharding: This process involves dividing data across multiple nodes in a distributed system, improving performance and scalability but increasing the complexity of maintaining data consistency.

ACID transactions are a fundamental concept in database management, providing benefits such as data integrity, consistency, isolation, and durability. They ensure that the database remains in a consistent state even if the server fails. However, they do have limitations and challenges, such as overhead, deadlocks, and scalability problems in distributed systems.

Although ACID transactions provide strong consistency and reliability, they may not always be the best option for every use case. Organizations must carefully evaluate their unique needs and requirements to determine whether ACID transactions or alternative transaction models, such as BASE or CAP, are a better fit for their systems.

CONSISTENT HASHING:

Hashing involves using a hash function to produce a pseudo-random number. This number is then divided by the size of the available memory space, resulting in the transformation of the random identifier into a position within the given memory space.

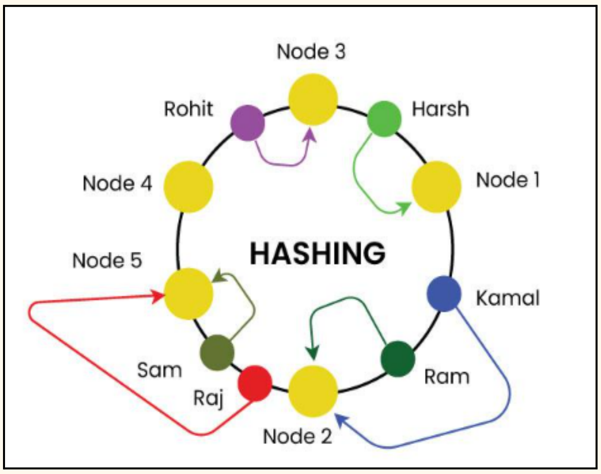


Consistent hashing is a technique used in computer systems to distribute keys (e.g., cache keys) uniformly across a cluster of nodes (e.g., cache servers). The goal is to minimize the number of keys that need to be moved when nodes are added or removed from the cluster, thus reducing the impact of these changes on the overall system.

- It represents the requests by the system/clients and the server nodes in a virtual ring structure which is known as a hashring.

- The number of locations in this ring is not fixed, but it is considered to have an infinite number of points
- The server nodes can be placed at random locations on this ring which can be done using hashing.
- The requests, that is, the users, computers, or serverless programs, are also placed on the same ring using the same hash function.

How does consistent hashing work?



1. The output of the hash function is placed on a virtual ring structure (known as the hash ring)
2. The hashed IP addresses of the nodes are used to assign a position for the nodes on the hash ring
3. The key of a data object is hashed using the same hash function to find the position of the key on the hash ring
4. The hash ring is traversed in the clockwise direction starting from the position of the key until a node is found
5. The data object is stored or retrieved from the node that was found

Phases/Working of Consistent Hashing:

1. **Hash Function Selection:** The first step in consistent hashing is to choose the hash function that will be used to associate keys with network nodes. For each key, this hash function ought to yield a different value and be deterministic. Keys will be consistently and predictably mapped to nodes using the chosen hash function.
2. **Node Assignment:** Based on the hash function's findings, nodes in the network are given keys in this phase. The nodes are organized in a circle, and the keys are given to the node that is situated closest to the key's hash value in a clockwise direction in the circle.
3. **Key Replication:** It's critical to make sure that data is accessible in a distributed system even in the case of node failures. Keys can be copied across a number of network nodes to accomplish this. In the event that one node fails, this helps to guarantee that data is always accessible.
4. **Node Addition/Removal:** In order to keep the system balanced as nodes are added to or removed from the network, it may be necessary to remap the keys to new nodes. Consistent hashing reduces the effect of new or removed nodes by merely remapping a small portion of keys to the new node.
5. **Load balancing:** Consistent hashing aids in distributing the load among the network's nodes. To keep the system balanced and effective when a node is overloaded, portions of its keys can be remapped to other nodes.
6. **Failure Recovery:** Keys assigned to a node can be remapped to other nodes in the network in the event of a node failure. This makes it possible to keep data current and constantly accessible, even in the event that a node fails.

Requirements:

Functional Requirements -

- Design an algorithm to horizontally scale the cache servers
- The algorithm must minimize the occurrence of hotspots in the network
- The algorithm must be able to handle internet-scale dynamic load
- The algorithm must reuse existing network protocols such as TCP/IP

Non-Functional Requirements -

- Scalable

- High availability
- Low latency
- Reliable

Implementation of Consistent Hashing algorithm:

1. Choose a Hash Function:
   a. Select a hash function that produces a uniformly distributed range of hash values. Common choices include MD5, SHA-1, or SHA-256.
2. Define the Hash Ring:
   a. Represent the range of hash values as a ring. This ring should cover the entire possible range of hash values and be evenly distributed.
3. Assign Nodes to the Ring:
   a. Assign each node in the system a position on the hash ring. This is typically done by hashing the node's identifier using the chosen hash function.
4. Key Mapping:
   a. When a key needs to be stored or retrieved, hash the key using the chosen hash function to obtain a hash value.
   b. Find the position on the hash ring where the hash value falls.
   c. Walk clockwise on the ring to find the first node encountered. This node becomes the owner of the key.
5. Node Additions:
   a. When a new node is added, compute its position on the hash ring using the hash function.
   b. Identify the range of keys that will be owned by the new node. This typically involves finding the predecessor node on the ring.
   c. Update the ring to include the new node and remap the affected keys to the new node.
6. Node Removals:
   a. When a node is removed, identify its position on the hash ring.
   b. Identify the range of keys that will be affected by the removal. This typically involves finding the successor node on the ring.
   c. Update the ring to exclude the removed node and remap the affected keys to the successor node.
7. Load Balancing:
   a. Periodically check the load on each node by monitoring the number of keys it owns.
   b. If there is an imbalance, consider redistributing some keys to achieve a more even distribution.

What is the asymptotic complexity of consistent hashing?

Operation	Time Complexity	Description
Add a node	O(k/n + logn)	O(k/n) for redistribution of keys O(logn) for binary search tree traversal
Remove a node	O(k/n + logn)	O(k/n) for redistribution of keys O(logn) for binary search tree traversal
Add a key	O(logn)	O(logn) for binary search tree traversal
Remove a key	O(logn)	O(logn) for binary search tree traversal

where k = total number of keys, n = total number of nodes [2, 7].

Advantages of using Consistent Hashing:

1. **Load balancing:** Consistent hashing helps to evenly distribute the network's workload among its nodes, preserving the system's effectiveness and responsiveness even as the amount of data increases and changes over time.
2. **Scalability:** Consistent hashing is extremely scalable, which means that it can adapt to changes in the number of nodes or the amount of data being processed with little to no influence on the performance of the entire system.
3. **Minimal Remapping:** Consistent hashing reduces the number of keys that must be remapped when a node is added or removed, ensuring that the system is robust and consistent even as the network changes over time.
4. **Increased Failure Tolerance:** Consistent hashing makes data always accessible and current, even in the case of node failures. The stability and dependability of the system as a whole are enhanced by the capacity to replicate keys across several nodes and remap them to different nodes in the event of failure.
5. **Simplified Operations:** The act of adding or removing nodes from the network is made easier by consistent hashing, which makes it simpler to administer and maintain a sizable distributed system.

Disadvantages of using Consistent Hashing:

1. **Hash Function Complexity:** The effectiveness of consistent hashing depends on the use of a suitable hash function. The hash function must produce a unique value for each key and be deterministic in order to be useful. The system's overall effectiveness and efficiency may be affected by how complicated the hash function is.
2. **Performance Cost:** The computing resources needed to map keys to nodes, replicate keys, and remap keys in the event of node additions or removals can result in some performance overhead when using consistent hashing.
3. **Lack of Flexibility:** In some circumstances, the system's ability to adapt to changing requirements or shifting network conditions may be constrained by the rigid limits of consistent hashing.
4. **High Resource Use:** As nodes are added to or deleted from the network, consistent hashing may occasionally result in high resource utilization. This can have an effect on the system's overall performance and efficacy.
5. **The complexity of Management:** Managing and maintaining a system that uses consistent hashing can be difficult and demanding, and it often calls for particular expertise and abilities.

RATE LIMITING:

Rate limiting is a technique to limit network traffic to prevent users from exhausting system resources. It imposes constraints on the frequency or volume of requests from clients to prevent overload, maintain stability, and ensure fair resource allocation.

- By setting limits on the number of requests allowed within a specific timeframe, rate limiting helps to mitigate the risk of system degradation, denial-of-service (DoS) attacks, and abuse of resources.
- APIs that use rate limiting can throttle or temporarily block any client that tries to make too many API calls. It might slow down a throttled user's requests for a specified time or deny them altogether. Rate limiting ensures that legitimate requests can reach the system and access information without impacting the overall application's performance.

- Rate limiting is commonly implemented in various contexts, such as web servers, APIs, network traffic management, and database access, to ensure optimal performance, reliability, and security of systems.

What is a Rate Limiter?

A rate limiter is a component that controls the rate of traffic or requests to a system. It is a specific implementation or tool used to enforce rate-limiting.

Why Is Rate Limiting Important?

- **Preventing Overload:** Rate limiting helps prevent overload situations by controlling the rate of incoming requests or actions. By imposing limits on the frequency or volume of requests, systems can avoid becoming overwhelmed and maintain stable performance.
- **Ensuring Stability:** By regulating the flow of traffic, rate limiting helps ensure system stability and prevents resource exhaustion. It allows systems to handle incoming requests in a controlled manner, avoiding spikes in demand that can lead to degradation or failure.
- **Mitigating DoS (denial-of-service) Attacks:** Rate limiting is an effective defense mechanism against denial-of-service (DoS) attacks, where attackers attempt to flood a system with excessive requests to disrupt its operation. By enforcing limits on request rates, systems can mitigate the impact of DoS attacks and maintain availability for legitimate users.
- **Fair Resource Allocation:** Rate limiting promotes fair resource allocation by ensuring that all users or clients have equitable access to system resources. By limiting the rate of requests, systems can prevent certain users from monopolizing resources and prioritize serving requests from a diverse user base.
- **Optimizing Performance:** Rate limiting can help optimize system performance by preventing excessive resource consumption and improving response times. By controlling the rate of incoming requests, systems can allocate resources more efficiently and provide better overall performance for users.
- **Protecting Against Abuse:** Rate limiting helps protect systems against abuse or misuse by limiting the frequency of requests from individual users or clients. It can prevent abusive behavior such as spamming, scraping, or unauthorized access, thereby safeguarding system integrity and security.

How Does Rate Limiting Work?

- Rate limiting works within applications, not in the web server.
- It typically involves tracking the IP addresses where requests originate and identifying the time lapsed between requests. IP addresses are the application's main way to identify who has made each request.
- Rate-limiting solutions work by measuring the elapsed time between every request from a given IP address and tracking the number of requests made in a set timeframe. If one IP address makes too many requests within the specified timeframe, the rate-limiting solution throttles the IP address and doesn't fulfill its requests for the next time frame.

Types of Rate Limits:

Administrators can define different parameters and methods and parameters when setting a rate limit. An organization's chosen rate-limiting technique depends on the objective and the required level of restriction. Here are three main approaches to rate limiting that an organization might implement:

1. **User rate limits —** It identifies the number of requests a given user makes, usually by tracking the user's IP address or API key. Users that exceed the specified rate limit will trigger the application to deny any further requests until the rate-limited timeframe resets. Alternatively, users can contact the developers to increase the rate limit.

2. **Geographic rate limits —** developers can further secure applications in a given geographic region by setting a rate limit for each specific region for a specified timeframe. For example, developers might predict that users in a given region will be less active between midnight and 9:00 am and define a lower rate limit for this timeframe. This approach helps prevent suspicious traffic and further reduces the risk of an attack.
3. **Server rate limits —** developers can set rate limits at the server level if they define a specific server to handle parts of an application. This approach provides more flexibility, allowing the developers to increase the rate limit on commonly used servers while decreasing the traffic limit on less active servers.

What Are the Algorithms Used for Rate Limiting?

1. **Fixed Window Rate Limiting:**
    - The fixed window counting algorithm tracks the number of requests within a fixed time window (e.g., one minute, one hour).
    - Requests exceeding a predefined threshold within the window are rejected or delayed until the window resets.
    - This algorithm provides a straightforward way to limit the rate of requests over short periods, but it may not handle bursts of traffic well.

    Example: Allow up to 100 requests per minute.
2. **Sliding Window Rate Limiting:**
    - The sliding window log algorithm maintains a log of timestamps for each request received.
    - Requests older than a predefined time interval are removed from the log, and new requests are added.
    - The rate of requests is calculated based on the number of requests within the sliding window.
    - This algorithm allows for more precise rate limiting and better handling of bursts of traffic compared to fixed window counting.

    Example: Allow up to 100 requests in any 60-second rolling window.
3. **Token Bucket Rate Limiting:**
    - The token bucket algorithm allocates tokens at a fixed rate into a “bucket.”
    - Each request consumes a token from the bucket, and requests are only allowed if there are sufficient tokens available.
    - Unused tokens are stored in the bucket, up to a maximum capacity.
    - This algorithm provides a simple and flexible way to control the rate of requests and smooth out bursts of traffic.

    Example: Allow up to 100 tokens per minute; each request consumes one token.
4. **Leaky Bucket Rate Limiting:**
    - The leaky bucket algorithm models a bucket with a leaky hole, where requests are added at a constant rate and leak out at a controlled rate.
    - Incoming requests are added to the bucket, and if the bucket exceeds a certain capacity, excess requests are either delayed or rejected.
    - This algorithm provides a way to enforce a maximum request rate while allowing some burstiness in traffic.

    Example: Allow up to 100 requests with a leak rate of 10 requests per second.
5. **Distributed Rate Limiting:**
    - Distributed rate limiting involves distributing rate limiting across multiple nodes or instances of a system to handle high traffic loads and improve scalability.
    - Techniques such as consistent hashing, token passing, or distributed caches are used to coordinate rate limiting across nodes.

    Example: Distribute rate limiting across multiple API gateways or load balancers.

6. **Adaptive Rate Limiting:**
    - Adaptive rate limiting adjusts the rate limits dynamically based on system load, traffic patterns, or other factors.
    - Machine learning algorithms, statistical analysis, or feedback loops may be used to adjust rate limits in real-time.

    Example: Automatically adjust rate limits based on server load or response times.

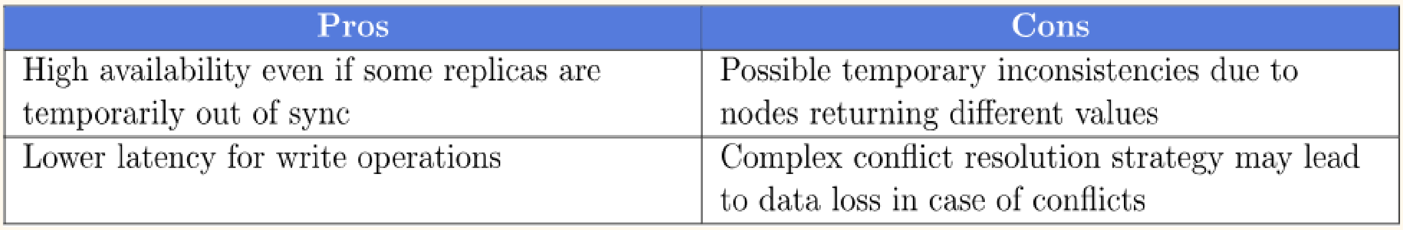
Rate Limiting in Different Layers of the System:

1. **Application Layer:**
    a. Rate limiting at the application layer involves implementing rate limiting logic within the application code itself. It applies to all requests processed by the application, regardless of their source or destination.
    b. Provides fine-grained control over rate limiting rules and allows for customization based on application-specific requirements.
    c. Useful for enforcing application-level rate limits, such as limiting the number of requests per user or per session.
2. **API Gateway Layer:**
    a. Rate limiting at the API gateway layer involves configuring rate limiting rules within the API gateway infrastructure. It applies to incoming requests received by the API gateway before they are forwarded to downstream services.
    b. Offers centralized control over rate limiting policies for all APIs and services exposed through the gateway.
    c. Suitable for enforcing global rate limits across multiple APIs, controlling access to public APIs, and protecting backend services from excessive traffic.
3. **Service Layer:**
    a. Rate limiting at the service layer involves implementing rate limiting logic within individual services or microservices. It applies to requests processed by each service independently, allowing for fine-grained control and customization.
    b. Each service can enforce rate limits tailored to its specific workload and resource requirements, enabling better scalability and resource utilization.
    c. Effective for controlling traffic to individual services, managing resource consumption, and preventing service degradation or failure due to overload.
4. **Database Layer:**
    a. Rate limiting at the database layer involves controlling the rate of database queries or transactions. It applies to database operations performed by the application or services, such as read and write operations.
    b. Helps prevent database overload and ensures efficient resource utilization by limiting the rate of database access.
    c. Useful for protecting the database from excessive load, preventing performance degradation, and ensuring fair resource allocation among multiple clients or services.

EVENTUAL VS STRONG CONSISTENCY IN DISTRIBUTED DATABASES:

Consistency Patterns - System Design

Eventual Consistency: Eventual consistency is a consistency model that enables the data store to be highly available. It is also known as optimistic replication & is key to distributed systems.



Pros	Cons
High availability even if some replicas are temporarily out of sync	Possible temporary inconsistencies due to nodes returning different values
Lower latency for write operations	Complex conflict resolution strategy may lead to data loss in case of conflicts

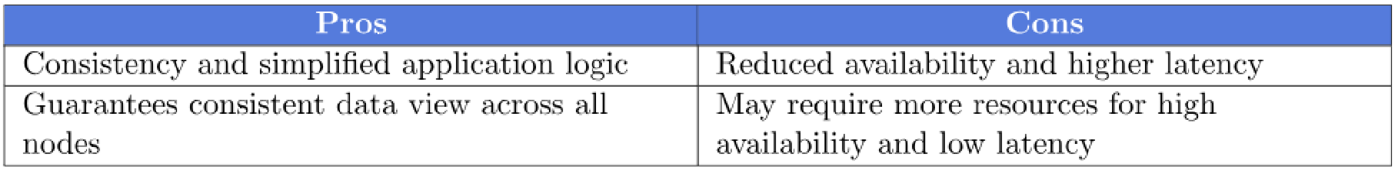
How exactly does it work?

- Think of a popular microblogging site deployed across the world in different geographical regions like Asia, America, and Europe. Moreover, each geographical region has multiple data center zones: North, East, West, and South.
- Furthermore, each zone has multiple clusters which have multiple server nodes running. So, we have many datastore nodes spread across the world that micro-blogging site uses for persisting data. Since there are so many nodes running, there is no single point of failure.
- The data store service is highly available. Even if a few nodes go down, persistence service is still up. Let’s say a celebrity makes a post on the website that everybody starts liking around the world.
- At a point in time, a user in Japan likes a post which increases the “Like” count of the post from say 100 to 101. At the same point in time, a user in America, in a different geographical zone, clicks on the post, and he sees “Like” count as 100, not 101.

Reason for the above Use case :

- Simply, because the new updated value of the Post “Like” counter needs some time to move from Japan to America and update server nodes running there. Though the value of the counter at that point in time was 101, the user in America sees old inconsistent values.
- But when he refreshes his web page after a few seconds, the “Like” counter value shows as 101. So, data was initially inconsistent but eventually got consistent across server nodes deployed around the world. This is what eventual consistency is.

Strong Consistency: Strong Consistency simply means the data must be strongly consistent at all times. All the server nodes across the world should contain the same value as an entity at any point in time. And the only way to implement this behavior is by locking down the nodes when being updated.



Pros	Cons
Consistency and simplified application logic	Reduced availability and higher latency
Guarantees consistent data view across all nodes	May require more resources for high availability and low latency

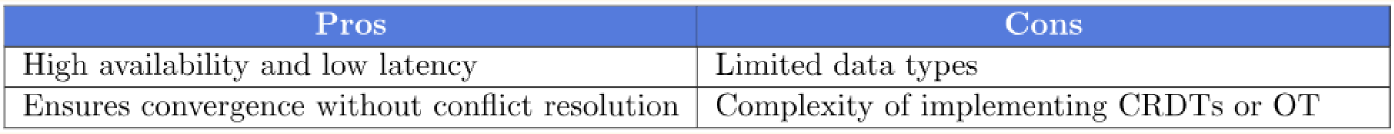
How exactly does it work?

- Let’s continue the same Eventual Consistency example. To ensure Strong Consistency in the system, when a user in Japan likes posts, all nodes across different geographical zones must be locked down to prevent any concurrent updates.
- This means at one point in time, only one user can update the post “Like” counter value. So, once a user in Japan updates the “Like” counter from 100 to 101. The value gets replicated globally across all nodes. Once all nodes reach consensus, locks get lifted. Now, other users can Like posts.
- If the nodes take a while to reach a consensus, they must wait until then. Well, this is surely not desired in the case of social applications. But think of a stock market application where the users are seeing different prices of the same stock at one point in time and updating it concurrently. This would create chaos. Therefore, to avoid this confusion we need our systems to be Strongly Consistent.

- The nodes must be locked down for updates. Queuing all requests is one good way of making a system Strongly Consistent. The strong Consistency model hits the capability of the system to be Highly Available & perform concurrent updates. This is how strongly consistent ACID transactions are implemented.

Strong Eventual Consistency (SEC):

- Strong eventual consistency guarantees that when all nodes have received the same set of updates, they'll be in the same state. Regardless of the order in which updates were applied. This is achieved through conflict-free replicated data types (CRDTs) or operational transformation (OT)
- SEC works only with specific data types that can be replicated and merged without conflicts. For example, SEC works well with data represented as a set or a counter since these data types can easily merge across nodes without conflicts. Riak, a distributed key-value store, supports strong eventual consistency through CRDTs.
- However, SEC may allow the system to have temporary inconsistencies. Often, these inconsistencies are likely to be caused by propagation delays rather than reconciliation conflicts.



Pros	Cons
High availability and low latency	Limited data types
Ensures convergence without conflict resolution	Complexity of implementing CRDTs or OT

SYNCHRONOUS VS ASYNCHRONOUS COMMUNICATION:

Synchronous Communication:

Synchronous communication refers to real-time interactions where participants exchange messages instantly.

Some key characteristics include:

- Live Back and Forth: With synchronous communication, recipients can provide immediate feedback, clarification and responses.
- Typically Scheduled: Most synchronous interactions like meetings and calls require advance planning and coordination.
- Happens in Real Time: All participants engage simultaneously without delays between message sending and receiving.
- Often In-Person: Traditional in-person conversations and meetings involve live synchronous communication.
- Examples of Synchronous Communication:
- In-Person Meetings and Events: In-person communication allows reading facial expressions and body language.
- Phone and Video Calls: Calls facilitate real-time discussion without being in the same physical place.
- Instant Messaging and Chat: Platforms like Slack enable live team conversations through text and video chat features.
- Virtual Conferences and Webinars: Remote events let speakers present and participants ask questions in real time.
- Classroom Learning: Physical and virtual classes enable students to actively discuss material and collaborate.

The Pros and Cons of Synchronous Communication:

Pros	Cons

Allows Immediate Feedback: Real-time interactions provide opportunities for instant reactions and clarification.	Risk of Multitasking and Distraction: It's easier for participants to get distracted and tune out during live interactions.
Builds Stronger Relationships: The conversational flow fosters deeper interpersonal connections.	Scheduling Challenges: Finding mutually suitable times for synchronous communication can be difficult, especially across time zones.
Enables Real-Time Problem Solving: Issues can be resolved quickly with back-and-forth dialogue.	Harder for Introverts: Some personality types feel more comfortable absorbing information vs. live interactions.

Asynchronous Communication:
Asynchronous communications happen with a lag between sending and receiving messages.

Some key characteristics include:

- Delay Between Correspondents: Recipients absorb information and craft responses on their own time.
- Typically Not Scheduled: Asynchronous exchanges like email don't require coordinated scheduling.
- Information Persists: Conversations remain documented for reference unlike ephemeral live chats.
- Enables Parallel Work: Teams can make progress in parallel without blocking each other.
- Examples of Asynchronous Communication:
- Email: Email allows flexible non-scheduled messages to individuals or groups.
- Online Discussion Boards: Forums and boards let communities exchange messages over time.
- Project Management Tools: Platforms like Trello, Asana and Basecamp enable task coordination through comments.
- Document Sharing and Wikis: Wikis and cloud docs provide venues for non-simultaneous editing and feedback.
- Recorded Video and Audio: Pre-recorded messages allow flexible consumption on viewers' own time.

The Pros and Cons of Asynchronous Communication:

Pros	Cons
Enables Flexible Scheduling: Participants can connect at their own convenience without live coordination.	Harder to Build Rapport: Lack of real-time interaction makes establishing closer connections more challenging.
Allows Time to Process Information: Recipients have space to absorb messages before thoughtfully responding.	Higher Risk of Miscommunication: Nuances around tone and intent get lost without face-to-face cues.
Persists Information: Conversations remain documented for future reference.	Delays in Resolving Issues: Problems can't be solved as quickly without live back-and-forth.

RPS vs REST:

What is it?

REST	RPC
Representational State Transfer (REST) is an architectural style that defines a set of constraints and protocols to be used for creating web services.	Remote Procedure Call (RPC) is a methodology used for constructing distributed, client-server-based applications. It is also called a subroutine call or a function call.
A REST API endpoint is a URL that utilizes HTTP verbs to execute CRUD (Create Read Update Delete) operations over the resources. These HTTP verbs are GET, POST, PATCH, PUT and DELETE. It focuses on providing resources from the server to the clients.	RPC is like calling a function. The client specifies the procedure to execute along with the parameters, and the server returns the results. This might appear more direct and granular compared to the resource-based approach. RPC only supports GET and POST requests.
1. They are very scalable as the client and server are decoupled easing to scale in the future. 2. Simple, standardized, and easy to use. 3. Uses already existing HTTP features. 4. They have high performance because of their cache capabilities. 5. Allows Standard-based protection with the use of OAuth protocols to verify REST requests. 6. Brings flexibility by serializing data in XML or JSON format.	1. They provide usage op applications in both local and distributed environments. 2. It provides ABSTRACTION. 3. They have lightweight payloads, therefore, provides high performance. 4. They are easy to understand and work as the action is part of the URL.
1. REST API's payload is quite big hence the entire files get back while you need one field. 2. It loses the ability to maintain state in REST.	1. It can be implemented in many ways as it is not well standardized. 2. It has no flexibility for hardware architecture.

How do they work?

Action	RPC	REST	Comment
Adding a new product to a product list	POST /addProduct HTTP/1.1 HOST: api.example.com Content-Type: application/json {"name": "T-Shirt", "price": "22.00", "category": "Clothes"}	POST /products HTTP/1.1 HOST: api.example.com Content-Type: application/json {"name": "T-Shirt", "price": "22.00", "category": "Clothes"}	RPC uses POST on function, and REST uses POST on URL.
Retrieving a product's details	POST /getProduct HTTP/1.1 HOST: api.example.com Content-Type: application/json {"productID": "123"}	GET /products/123 HTTP/1.1 HOST: api.example.com	RPC uses POST on function and passes parameter as JSON object. REST uses GET on URL and

			passes parameter in URL.
Updating a product’s price	POST /updateProductPrice HTTP/1.1 HOST: api.example.com Content-Type: application/json {"productId": "123", "newPrice": "20.00"}	PUT /products/123 HTTP/1.1 HOST: api.example.com Content-Type: application/json {"price": "20.00"}	RPC uses POST on function and passes parameter as JSON object. REST uses PUT on URL and passes parameter in URL and as JSON object.
Deleting a product	POST /deleteProduct HTTP/1.1 HOST: api.example.com Content-Type: application/json {"productId": "123""}	DELETE /products/123 HTTP/1.1 HOST: api.example.com	RPC uses POST on function and passes parameter as JSON object. REST uses DELETE on URL and passes parameter in URL.

When to use:

Here are actions where RPC is a good option:

1. Take a picture with a remote device’s camera
2. Use a machine learning algorithm on the server to identify fraud
3. Transfer money from one account to another on a remote banking system
4. Restart a server remotely

Here are actions where a REST API is a good option:

1. Add a product to a database
2. Retrieve the contents of a music playlist
3. Update a person’s address
4. Delete a blog post

	RPC	**REST**
What is it?	A system allows a remote client to call a procedure on a server as if it were local. RPC is action-oriented.	A set of rules that defines structured data exchange between a client and a server. REST is resource-oriented.
Used for	Performing actions on a remote server.	Create, read, update, and delete (CRUD) operations on remote objects.
Best fit	When requiring complex calculations or triggering a remote process on the server. RPC only supports GET and POST requests.	When server data and data structures need to be exposed uniformly. Supports HTTP methods GET, POST, PUT, PATCH, and DELETE.
Statefulness	Stateless or stateful.	Stateless.
Data passing format	In a consistent structure defined by the server and enforced on the client. Require payloads of a few data types as XML for XML-RPC.	In a structure determined independently by the server. Multiple different formats can be passed within the same API.

		REST allows to specify Content-types or accept headers

BATCH PROCESSING vs STREAM PROCESSING:

Batch Processing:

- It is a data processing technique where data is collected, processed, and stored in predefined chunks or batches over a period of time.
- Data is segmented into specific blocks or chunks, and each batch is processed sequentially. There's often a start and end to each batch.
- Since data is processed in chunks after collection, it's often complete and consistent, reducing the chances of missing data.
- It has higher latency since data is not processed immediately. It waits for a batch to be complete or a specific schedule to trigger the processing.
- If a batch processing job fails, it can be restarted from where it left off, or the entire batch can be reprocessed. So it provides fault tolerance mechanisms, ensuring data integrity and allowing for the recovery of failed jobs.
- Systems are often optimized for throughput since large volumes of data are processed at once. They might be scaled vertically (more powerful machines) or horizontally (more machines) depending on the use case.
- Might have a simpler setup and design since it doesn't always need to account for real-time processing complexities.

Stream Processing:

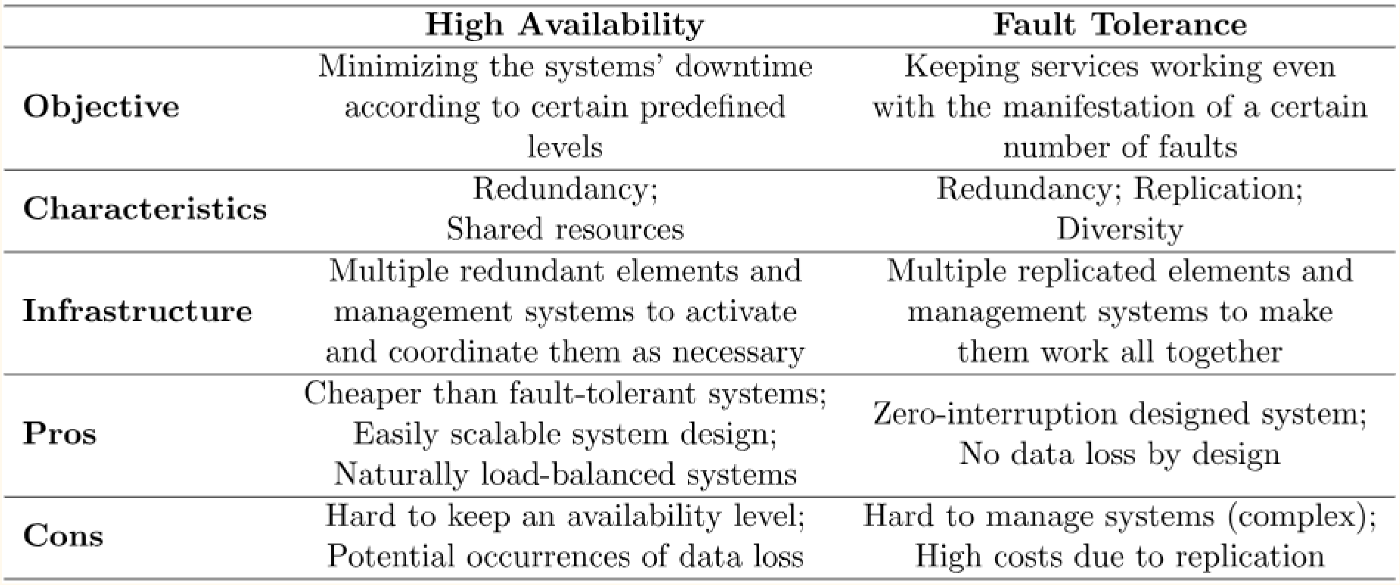
- It is designed to process data in real-time or near-real-time. As soon as the data arrives, it's processed, which means there's no waiting for a batch of data to accumulate.
- Data is continuous and unbounded. Processing involves handling infinite data streams, with no predefined start or end.
- As data is processed in real-time, there's a chance for out-of-order data or potential gaps, requiring mechanisms to handle such inconsistencies.
- It has lower latency because data is processed immediately as it flows into the system, which makes it more suitable for real-time analytics or tasks requiring instantaneous insights.
- Requires more sophisticated fault tolerance mechanisms. If a data stream is interrupted, the system needs ways to handle the interruption and ensure data isn't lost.
- Systems need to be designed for both high throughput and low latency. They're usually scaled horizontally to handle varying data velocities.
- Often requires a more complex setup, especially when ensuring fault tolerance, managing state, and dealing with out-of-order data events.

Criteria	Batch Processing	Stream Processing
Nature of Data	Processed in chunks or batches & data size is known and finite.	Processed continuously, one event at a time & data size is unknown and infinite in advance.
Latency	High latency: insights are obtained after the entire batch is processed.	Low latency: insights are available almost immediately or in near-real-time.
Processing Time	Scheduled (e.g., daily, weekly).	Continuous.

Infrastructure Needs	Significant resources might be required but can be provisioned less frequently.	Requires systems to be always on and resilient.
Throughput	High: can handle vast amounts of data at once.	Varies: optimized for real-time but might handle less data volume at a given time.
Complexity	Relatively simpler as it deals with finite data chunks.	More complex due to continuous data flow and potential order or consistency issues.
Ideal Use Cases	Data backups, ETL jobs, monthly reports.	Real-time analytics, fraud detection, live dashboards.
Error Handling	Detected after processing the batch; might need to re-process data.	Needs immediate error-handling mechanisms; might also involve later corrections.
Consistency & Completeness	Data is typically complete and consistent when processed.	Potential for out-of-order data or missing data points.
Tools & Technologies	Hadoop, Apache Hive, batch-oriented Apache Spark.	Apache Kafka, Apache Flink, Apache Storm.

FAULT TOLERANCE vs HIGH AVAILABILITY:

- Fault Tolerance refers to a system's capacity to sustain its functionality in the presence of hardware or software failures.
- It involves implementing redundancy, error detection, and error recovery mechanisms to ensure that the system can continue to operate or degrade at a lesser rate in performance rather than experiencing a catastrophic failure.



	High Availability	Fault Tolerance
Objective	Minimizing the systems' downtime according to certain predefined levels	Keeping services working even with the manifestation of a certain number of faults
Characteristics	Redundancy; Shared resources	Redundancy; Replication; Diversity
Infrastructure	Multiple redundant elements and management systems to activate and coordinate them as necessary	Multiple replicated elements and management systems to make them work all together
Pros	Cheaper than fault-tolerant systems; Easily scalable system design; Naturally load-balanced systems	Zero-interruption designed system; No data loss by design
Cons	Hard to keep an availability level; Potential occurrences of data loss	Hard to manage systems (complex); High costs due to replication

VERTICAL SCALING vs HORIZONTAL SCALING:

Point of comparison	Horizontal scaling (scaling out)	Vertical scaling (scaling up)

How the scaling works	You add new same-size servers to an existing pool of machines	You upgrade components of an existing server (or get a new device to substitute the current one)
Process complexity	High (requires load balancing and code for managing data consistency)	Low (turn off the server, take out the old component, set up the new one, and restart the device)
Message passing	Horizontal scaling involves distributed computing that lacks a shared address space. This means that the machines need to communicate with one another and exchange data within the framework. As this involves copies of information, it can be costly.	Vertical scaling includes a shared address space, where all computing resources exist within a system or server. This makes data sharing and message passing easier and less complicated.
Concurrency	Distributes multiple jobs across multiple machines over the network, at a go. This reduces the workload on each machine	Relies on multi-threading on the existing machine to handle multiple requests at the same time
Overall cost	High (you purchase new servers every time you scale)	Low (if you're buying only one or two new components)
Load balancing	Yes. Necessary to actively distribute workload across the multiple nodes.	No
Single point of failure	No because other machines in the cluster offer backup.	Yes since it's a single source of failure.
Performance concerns	Nodes communicate over network calls (RPC), which slows down system performance	Everything runs on the same server, which boosts performance
Data storage	You distribute data across multiple nodes	All data lives on the same server
Data consistency	An issue since data moves between different nodes	Data resides on one system, so there's less chance for dirty reads or dirty writes
An upper limit for scalability	No (you can always add more machines)	Yes (each machine has a threshold for RAM, storage, and processing power)
Downtime during scaling	No	Yes

Rapid growth handling	Highly flexible (even automatic in some cases)	Manual and inflexible
Risk of unused resources	Low	High
Required code reworks	Requires breaking a sequential piece of logic so that workloads run in parallel across multiple machines	The logic does not change (you just run the same code on a higher-spec device)
Ability to scale down	Yes	Not without taking the machine offline and removing components
Database programs	Cassandra, Google Cloud Spanner, and MongoDB	MySQL and Amazon RDS

When to Choose Vertical vs Horizontal scaling:

Things to consider to decide between vertical and horizontal scaling:

- **Cost:** Analyze initial hardware costs vs. long-term operational expenses.
- **Workload:** Is your application CPU bound, memory bound, or does it lend itself to distribution?
- **Architectural Complexity:** Can your application code handle distributed workloads?
- **Future Growth:** How much scaling do you realistically anticipate?

When to Choose Vertical Scaling:

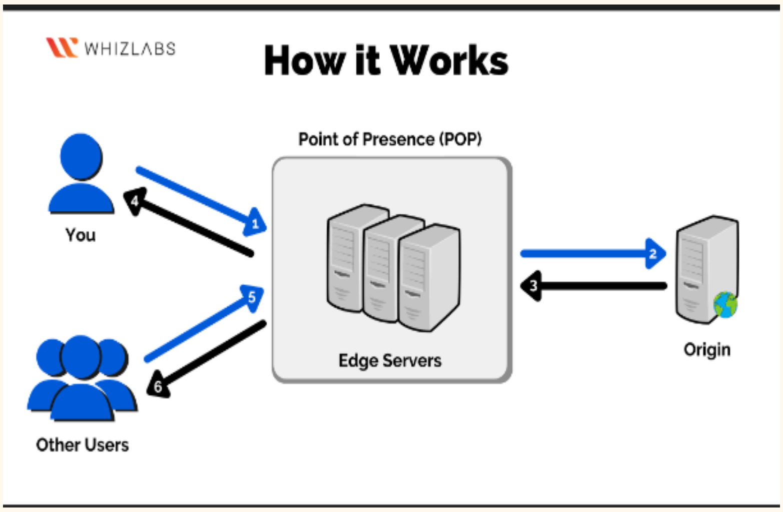
1. **Limited Scalability:** Small to medium-sized applications with a limited growth forecast and your needs are easily met by hardware upgrades.
2. **Legacy applications:** When there is a tight coupling between components, making it difficult to distribute across multiple servers.
3. **Low Latency:** When low latency is a critical requirement, and inter-server communication overhead is unacceptable.
4. **Cost-sensitive projects:** When the budget does not allow for a complex infrastructure and the cost of scaling horizontally outweighs the benefits, such as in the case of expensive software licenses.

When to Choose Horizontal Scaling:

1. **Rapid Growth:** When experiencing rapid growth and requiring the ability to handle increasing traffic.
2. **High availability needs:** When the application needs to be highly available and resilient to node failures.
3. **Easily Distributable:** When the application can be easily distributed across multiple servers without significant modifications.
4. **Microservices architectures:** When applications are designed around microservices, which naturally lend themselves to horizontal scaling.
5. **Cost Effectiveness:** When cost-effectiveness is a priority, and the use of commodity hardware is preferred.

CONTENT DELIVERY NETWORK(CDN):

A Content Delivery Network (CDN) is a system of distributed servers (network) that delivers web content to users based on their geographic location, the origin of the web page, and a content delivery server. CDNs are used to deliver content such as HTML pages, JavaScript files, stylesheets, images, and videos more quickly and efficiently.



Key Components and Features of CDNs:

1. Distributed Servers:
   1.1. Edge Servers: Located close to the end-users to serve content quickly.
   1.2. Origin Servers: The central repository of content.
2. Caching: Static Content: Files like images, CSS, and JavaScript that do not change frequently.
3. Dynamic Content: Content that changes frequently and may require additional logic to cache effectively.
4. Geographic Distribution: Servers are placed strategically around the globe to minimize latency and improve load times by serving content from the nearest edge server.
5. Load Balancing: Distributes the load across multiple servers to ensure no single server becomes a bottleneck, enhancing reliability and performance.
6. Content Optimization: Compresses files, minifies scripts, and uses other techniques to reduce the size of files delivered to the end-user.
7. Security: Provides DDoS protection, secure token authentication, and SSL/TLS encryption to protect content and user data.

How CDNs Work:

1. User Request: A user requests a webpage or file from a website.
2. DNS Resolution: The CDN's DNS system redirects the request to the nearest edge server.
3. Edge Server Response: The edge server checks its cache for the requested content. If the content is cached (cache hit), it is delivered to the user. If the content is not cached (cache miss), the edge server retrieves it from the origin server, caches it, and then delivers it to the user.

Benefits of Using a CDN:

1. Improved Load Times: By serving content from the nearest edge server, CDNs reduce latency and load times.
2. Scalability: CDNs handle high traffic volumes and sudden spikes efficiently, ensuring consistent performance.
3. Reliability and Redundancy: With multiple servers distributed globally, CDNs provide redundancy, reducing the risk of downtime.
4. Reduced Bandwidth Costs: By caching content and reducing the load on origin servers, CDNs help lower bandwidth consumption and costs.
5. Enhanced Security: CDNs offer various security features to protect against threats and ensure safe content delivery.

Examples of Popular CDNs:

- Akamai: One of the largest and most established CDNs, known for its extensive network and robust security features.
- Cloudflare: Offers a free tier and integrates security features like DDoS protection and a web application firewall (WAF).
- Amazon CloudFront: Part of the AWS ecosystem, offering seamless integration with other AWS services.
- Fastly: Known for real-time content delivery and edge computing capabilities.

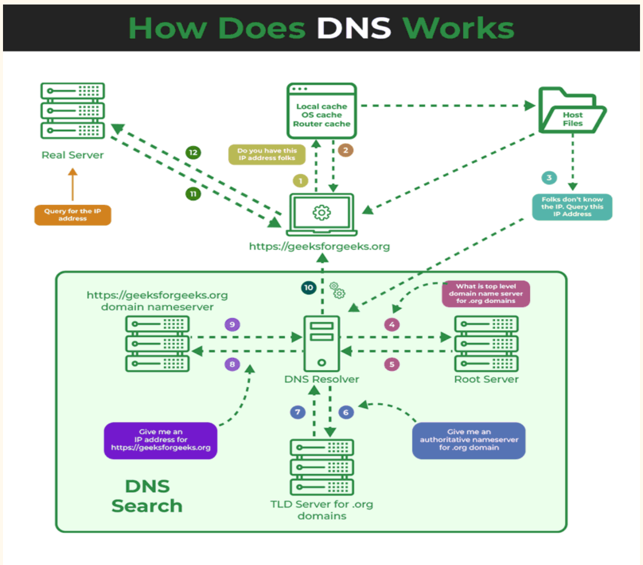
- Google Cloud CDN: Integrated with Google Cloud services, providing fast and secure content delivery.

DOMAIN NAME SYSTEM (DNS):

The domain name system (DNS) is a naming database in which internet domain names are located and translated into Internet Protocol (IP) addresses. The domain name system maps the name people use to locate a website to the IP address that a computer uses to locate that website.

- **Purpose:**
  - Every host is identified by the IP address but remembering numbers is very difficult for people. Also the IP addresses are not static therefore a mapping is required to change the domain name to the IP address. So DNS is used to convert the domain name of the websites to their numerical IP address.
  - DNS provides a mapping between human-friendly domain names (like google.com) and their corresponding IP addresses (like 172.217.164.142). It ensures that when you type a domain name in your browser, it resolves to the correct IP address.
- **Hierarchy:** DNS operates in a hierarchical manner. It consists of multiple servers, organized into zones. The top-level domain servers (like .com, .org, etc.) handle requests for specific domain extensions. Below them are authoritative servers for individual domains.
- **Types of Domains:**
  - **Generic Domains:** These define registered hosts based on their behavior. Examples include .com (commercial), .edu (educational institutions), and .gov (government).
  - **Country Domains:** These use two-character country abbreviations (e.g., .us for the United States) instead of organizational abbreviations.
  - **Inverse Domain:** Used for mapping IP addresses back to domain names.
- **Working of DNS:**
  - When you enter a domain name in your browser, a DNS resolver (usually provided by your ISP) sends a request to a DNS server.
  - The DNS server looks up the IP address associated with the domain name in its database.
  - If found, it returns the IP address to the resolver, which then connects you to the correct server.

How does DNS work?


How Does DNS Works
Real Server
Query for the IP address
Local cache
OS cache
Router cache
Do you have this IP address folks
Host Files
Folks don't know the IP. Query this IP Address
https://geeksforgeeks.org
1
2
3
11
12
https://geeksforgeeks.org domain nameserver
10
What is top level domain name server for .org domains
9
8
4
5
DNS Resolver
Root Server
7
6
Give me an IP address for https://geeksforgeeks.org
Give me an authoritative nameserver for .org domain
DNS Search
TLD Server for .org domains

DNS servers convert URLs and domain names into IP addresses that computers can understand and use. They translate what a user types into a browser into something the machine can use to find a webpage. This process of translation and lookup is called DNS resolution.

The basic process of a DNS resolution follows these steps:

1. The user enters a web address or domain name into a browser.
2. The browser sends a message, called a recursive DNS query to the network to find out which IP or network address the domain corresponds to.
3. The query goes to a recursive DNS server, which is also called a recursive resolver, and is usually managed by the internet service

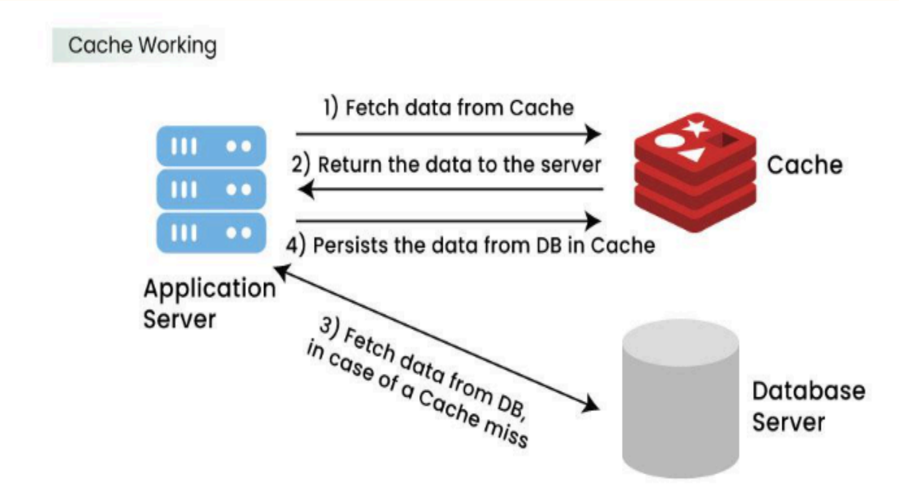
provider (ISP). If the recursive resolver has the address, it will return the address to the user, and the webpage will load.

4. If the recursive DNS server does not have an answer, it will query a series of other servers in the following order: DNS root name servers, top-level domain (TLD) name servers and authoritative name servers.
5. The three server types work together and continue redirecting until they retrieve a DNS record that contains the queried IP address. It sends this information to the recursive DNS server, and the webpage the user is looking for loads. DNS root name servers and TLD servers primarily redirect queries and rarely provide the resolution themselves.
6. The recursive server stores, or caches, the A record for the domain name, which contains the IP address. The next time it receives a request for that domain name, it can respond directly to the user instead of querying other servers.
7. If the query reaches the authoritative server and it cannot find the information, it returns an error message.

The entire process querying the various servers takes a fraction of a second and is usually imperceptible to the user. DNS servers answer questions from both inside and outside their own domains. When a server receives a request from outside the domain for information about a name or address inside the domain, it provides the authoritative answer. When a server gets a request from within its domain for a name or address outside that domain, it forwards the request to another server, usually one managed by its ISP.

CACHING:

Caching is a technique used to store and retrieve frequently accessed data or computations to speed up subsequent data requests. By storing data temporarily in a cache, systems can reduce the time and resources required to fetch the same data from its original source, leading to improved performance and reduced latency.


Cache Working
1) Fetch data from Cache
2) Return the data to the server
Cache
4) Persists the data from DB in Cache
Application Server
3) Fetch data from DB, in case of a Cache miss
Database Server

- **How it works:**
    - When data is requested, the system first checks if the data is stored in the cache.
    - If it is, the system retrieves the data from the cache rather than from the original source.
    - This can significantly reduce the time it takes to access the data.
- **Types of caching:**
    - In-memory caching stores data in memory, while disk caching stores data on a local hard drive.
    - Local caching refers to storing data on a single machine or within a single application. It's commonly used in scenarios where data retrieval is limited to one machine or where the volume of data is relatively small. Examples of local caching include browser caches or application-level caches.
    - Distributed caching involves storing data across multiple machines or nodes, often in a network. This type of caching is essential for applications that need to scale across multiple servers or are distributed geographically. Distributed caching ensures that data is available close to where it's needed, even if the original data source is remote or under heavy load.

- **Cache eviction:**
    - Caches can become full over time, which can cause performance issues.
    - To prevent this, caches are typically designed to automatically evict older or less frequently accessed data to make room for new data.
- **Cache consistency:**
    - Caching can introduce issues with data consistency, particularly in systems where multiple users or applications are accessing the same data.
    - To prevent this, systems may use cache invalidation techniques or implement a cache consistency protocol to ensure that data remains consistent across all users and applications.

Caching Policies:

- **Cache Writing Policies:** These policies determine when the application should write data to the cache.
    - **Write-Through:** The cache and the database are updated at the same time. This will ensure the cache and the database have the same data all the time but will slow down the write operation since it has to wait for both writes to succeed.
    - **Write-Back/Write-Behind:** Data is first written to cache and the write to the database is done later. The disadvantage is the potential inconsistency between cache and database since writing to database happens asynchronously.
    - **Write-Around:** Data is written directly to permanent storage, bypassing the cache. This can reduce the cache being flooded with write I/O that will not subsequently be re-read, but has the disadvantage that a read request for recently written data will create a “cache miss” and must be read from slower back-end storage and experience higher latency.
- **Cache Reading Policies:** These policies decide how to read data.
    - **Cache-Aside/Lazy Loading:** Data is loaded into the cache on demand, i.e., only when a cache miss occurs.
    - **Read-Through:** The cache takes responsibility for reading data from the database if it doesn't have the data.
- **Cache Eviction Policies:** These policies decide which item should be removed from the cache when the cache is full and a new item needs to be added. LRU and TTL fall into this category.
    - **LRU:** Discards the least recently used items first.
    - **TTL:** Data is kept or removed based on a time limit.

Cache Invalidation Strategies:

Cache invalidation is crucial in systems that use caching to enhance performance. When data is cached, it’s stored temporarily for quicker access. However, if the original data changes, the cached version becomes outdated. Cache invalidation mechanisms ensure that outdated entries are refreshed or removed, guaranteeing that users receive up-to-date information.

- Common strategies include time-based expiration, where cached data is discarded after a certain time, and event-driven invalidation, triggered by changes to the underlying data.
- Proper cache invalidation optimizes performance and avoids serving users with obsolete or inaccurate content from the cache.

Eviction Policies of Caching:

Eviction policies are crucial in caching systems to manage limited cache space efficiently. When the cache is full and a new item needs to be stored, an eviction policy determines which existing item to remove.

- One common approach is the Least Recently Used (LRU) policy, which discards the least recently accessed item. This assumes that recently used items are more likely to be used again soon.
- Another method is the Least Frequently Used (LFU) policy, removing the least frequently accessed items.
- Alternatively, there’s the First-In-First-Out (FIFO) policy, evicting the oldest cached item.

Advantages of using Caching:

- Caching optimizes resource usage, reduces server loads, and enhances overall scalability, making it a valuable tool in software development.
- Improved performance: Caching can significantly reduce the time it takes to access frequently accessed data, which can improve system performance and responsiveness.
- Reduced load on the original source: By reducing the amount of data that needs to be accessed from the original source, caching can reduce the load on the source and improve its scalability and reliability.
- Cost savings: Caching can reduce the need for expensive hardware or infrastructure upgrades by improving the efficiency of existing resources.

Disadvantages of using Caching:

- Despite its advantages, caching comes with drawbacks also and some of them are:
- Data inconsistency: If cache consistency is not maintained properly, caching can introduce issues with data consistency.
- Cache eviction issues: If cache eviction policies are not designed properly, caching can result in performance issues or data loss.

Feature	Redis	Memcached
Data Structures	Strings, lists, sets, sorted sets, hashes, bitmaps	Key-value pairs (strings or binary data)
Persistence	Optional (can save data to disk and recover after restart)	No (in-memory only)
Atomic Operations	Yes (e.g., increments, list manipulations)	No
Pub/Sub	Yes (built-in publish/subscribe messaging system)	No
High Availability	Yes (with Redis Sentinel and Redis Cluster)	No (third-party solutions available)
Complexity	More features and data structures, more complex	Simple and straightforward
Cache Eviction Policy	Configurable (e.g., LRU, LFU, volatile, or all keys)	Least Recently Used (LRU)
Use Cases	Advanced data structures, real-time applications, complex caching etc	Simple caching, session storage
Companies Using	Twitter, GitHub, Stack Overflow	Facebook, YouTube, Reddit

LOAD BALANCING:

Load balancing refers to efficiently managing traffic across a set of servers, also known as server farms or server pools. Each load balancer sits between client devices and backend servers, receiving and then distributing incoming requests to any available server capable of fulfilling them.

Load Balancer:

A load balancer is a networking device or software application that distributes and balances the incoming traffic among the servers to provide high availability, efficient utilization of servers, and high performance.

- Load balancers are highly used in cloud computing domains, data centers, and large-scale web applications where traffic flow needs to be managed.
- The primary goal of using a load balancer is, not to overburden with huge incoming traffic which may lead to server crashes or high latency.
- Load Balancers are categorized into two different groups: Layer 4 and Layer 7. Layer 4 corresponds to load balancing on a network level (Network & Transport Layer), optimizing the flow of packets through protocols like IP, TCP, FTP etc. Layer 7 works on an application level, optimizing HTTP requests, APIs etc.

How does a Load Balancer work:

1. **Client Request:** A client sends a request to access a web application or service.
2. **DNS Resolution:** The DNS server resolves the application's domain name to the IP address of the load balancer.
3. **Traffic Distribution:** The load balancer receives the request and applies the configured algorithm to select a backend server.
4. **Health Checks:** Before forwarding the request, the load balancer checks the health of the servers using predefined health checks (e.g., ping, HTTP status codes).
5. **Forwarding Request:** The load balancer forwards the client's request to the selected server.
6. **Server Response:** The server processes the request and sends a response back to the load balancer.
7. **Response to Client:** The load balancer forwards the server's response to the client.

Layer 4 Load Balancing vs. Layer 7 Load Balancing:

Feature	Layer-4 Load Balancing	Layer-7 Load Balancing
Operating Level:	Layer-4 Load Balancer Works at the Transport layer (OSI model) its task is analyzing basic information like IP addresses, ports, and TCP/UDP protocols. It's like a traffic cop directing cars based on lanes and signs.	Operates at the Application layer hence is responsible for examining deeper details like HTTP headers, URLs, and even content. Think of it as a smart assistant who reads the destination on each package and routes accordingly.
Decision Making:	Makes quick decisions based on simple metrics like server load or response times	Analyzes more complex data allowing for smarter routing based on specific needs and hence take generally more time in Decision making

Performance:	Faster due to its lightweight nature as it doesn't have to delve into packet content so it's Ideal for high-volume traffic with basic requirements	It generally has slightly slower processing due to content inspection but it offers the fine-grained control that outweighs the speed difference.
Cost:	Generally less expensive due to its simpler technology	More expensive due to its advanced features and processing power.
Additional Features:	Limited features beyond basic load balancing.	Offers advanced features like content caching, security filtering, and application health checks, providing more control and security.
Choosing the Right Fit:	Ideal for high-performance scenarios with basic traffics like distributing database queries or balancing generic web traffic.	Perfect for complex applications that require intelligent routing based on user data, content type, or specific server capabilities.

When to use Layer-4 Load Balancing over Layer-7 Load Balancing?

- Performance is paramount: Layer-4 operates at the transport layer, making decisions solely based on IP addresses and ports, resulting in faster processing and lower latency. Ideal for high-traffic scenarios and applications focused on raw speed, like DNS, video streaming, and gaming servers.
- Simplicity is key: Layer-4 uses simpler algorithms and requires less computational power, making it easier to manage and deploy. Often preferred for basic load balancing needs without complex requirements.
- Cost is a concern: Layer-4 hardware and software tend to be less expensive due to their simpler nature. Can be a cost-effective choice for basic load balancing needs.

When to use Layer-7 Load Balancing over Layer-4 Load Balancing?

- Application Awareness is Needed: Layer-7 load balancers have deep visibility into application traffic and can make intelligent load balancing decisions based on application-specific criteria.
- Content-based Routing is Required: Layer-7 load balancers can route requests to specific backend servers based on the content of the request, such as URLs, HTTP headers, or message payloads. This enables sophisticated routing strategies.
- Session Persistence is Necessary: Layer-7 load balancers can maintain session affinity or sticky sessions by associating client requests with specific backend servers based on session identifiers, cookies, or other application-level attributes.

SQL vs NoSQL:

What is SQL?

SQL is a domain-specific language used to query and manage data. It works by allowing users to query, insert, delete, and update records in relational databases. SQL also allows for complex logic to be applied through the use of transactions and embedded procedures such as stored functions or views.

What is NoSQL?

NoSQL stands for Not only SQL. It is a type of database that uses non-relational data structures, such as documents, graph databases, and key-value stores to store and retrieve data. NoSQL systems are designed to be more flexible than traditional relational databases and can scale up or down easily to accommodate changes in usage or load. This makes them ideal for use in applications.

SQL	NoSQL
Databases are categorized as Relational Database Management System (RDBMS).	NoSQL databases are categorized as Non-relational or distributed database system.
SQL databases have fixed or static or predefined schema.	NoSQL databases have dynamic schema.
SQL databases display data in the form of tables so it is known as table-based databases.	NoSQL databases display data as collections of key-value pairs, documents, graph databases or wide-column stores.
SQL databases are vertically scalable.	NoSQL databases are horizontally scalable.
SQL databases are best suited for complex queries	NoSQL databases are not so good for complex queries because these are not as powerful as SQL queries.
SQL databases use a powerful language "Structured Query Language" to define and manipulate the data.	In NoSQL databases, collections of documents are used to query the data. It is also called unstructured query language. It varies from database to database.
SQL databases are not best suited for hierarchical data storage.	NoSQL databases are best suited for hierarchical data storage.
Follows ACID property.	Follows CAP(consistency, availability, partition tolerance)
MySQL, Oracle, Sqlite, PostgreSQL and MS-SQL etc. are examples of SQL databases.	MongoDB, BigTable, Redis, RavenDB, Cassandra, Hbase, Neo4j, CouchDB etc. are examples of NoSQL databases.

When to use: SQL vs NoSQL?

SQL is a good choice when working with related data. Relational databases are efficient, flexible, and easily accessed by any application. A benefit of a relational database is that when one user updates a specific record, every instance of the database automatically refreshes, and that information is provided in real-time.

SQL and a relational database make it easy to handle a great deal of information, scale as necessary and allow flexible access to data only needing to update data once instead of changing multiple files, for instance. It’s also best for assessing data integrity. Since each piece of information is stored in a single place, there’s no problem with former versions confusing the picture.

While NoSQL is good when the availability of big data is more crucial, SQL is valued for ensuring data validity. It’s also a wise decision when a business needs to expand in response to shifting customer demands. NoSQL offers high performance, flexibility, and ease of use.

NoSQL is also a wise choice when dealing with large or constantly changing data sets, flexible data models, or requirements that don't fit into a relational model. Document databases, like CouchDB, MongoDB, and Amazon DocumentDB, are useful for handling large amounts of unstructured data.

Which is better: SQL or NoSQL?

The decision of which type of database to use - SQL or NoSQL - will depend on the particular needs and requirements of the project. For example, if you need a fast, scalable, and reliable database for web applications then a NoSQL system may be preferable. On the other hand, if your application requires complex data queries and transactional support then an SQL system may be the better choice. Ultimately, there is no one-size-fits-all solution - it all comes down to what you need from your database and which type of system can provide that in the most efficient manner. It's best to research both options thoroughly before making a decision.

How do I choose between a SQL and NoSQL database?

Choosing between SQL and NoSQL databases depends on the workload you plan to support and the structure and amount of data you have. SQL databases are a better option for applications that require multi-row transactions such as an accounting system or for legacy systems that were built for a relational structure. On the other hand, NoSQL databases are preferred for large or ever-changing data sets because they are horizontally scalable, meaning that you can handle more traffic by sharding or adding more servers in your NoSQL database. SQL databases, on the other hand, are vertically scalable, meaning that you can increase the load on a single server by increasing things like RAM, CPU, or SSD. SQL databases are table-based, whereas NoSQL databases are either key-value pairs, document-based, graph databases, or wide-column stores. SQL databases follow ACID properties (Atomicity, Consistency, Isolation, and Durability), whereas NoSQL databases follow the Brewers CAP theorem (Consistency, Availability, and Partition tolerance).

If you're still unsure which type of database to choose, you can consider the following factors:

1. **Data structure:** If your data is structured and predictable, SQL databases are a better option. If your data is unstructured and dynamic, NoSQL databases are a better option.
2. **Scalability:** If you need to scale horizontally, NoSQL databases are a better option. If you need to scale vertically, SQL databases are a better option.
3. **Data volume:** If you have a large amount of data, NoSQL databases are a better option. If you have a small amount of data, SQL databases are a better option.
4. **Query complexity:** If you have complex queries, SQL databases are a better option. If you have simple queries, NoSQL databases are a better option.

Can SQL databases handle the same types of data as NoSQL databases?

SQL databases are designed for structured data and are highly effective in managing complex queries and transactions with relational data. While they can handle structured data efficiently, they are not as flexible as NoSQL databases in dealing with unstructured or semi-structured data due to their fixed schema requirements.

Strategies for migrating from SQL to NoSQL: Strategies For Migrating From SQL to NoSQL Database - GeeksforGeeks

Here are some General Strategies -

1. Understand the differences between SQL and NoSQL databases – It's important to understand the differences between SQL and NoSQL databases in order to make an informed decision about which type of database is best suited for your needs. NoSQL databases are generally better suited for handling large amounts of unstructured data, while SQL databases are better suited for structured data with complex relationships.

2. Identify the use cases for the NoSQL database – Determine the specific use cases for which you will be using the NoSQL database. This will help you select the appropriate NoSQL database and design the schema in a way that is optimized for your use cases.
3. Evaluate the data migration options – There are several different approaches to migrating data from a SQL to a NoSQL database, including writing custom scripts, using ETL tools, or using a managed service. Evaluate the pros and cons of each approach to determine the best fit for your needs.
4. Test and validate the migrated data – It's important to thoroughly test and validate the migrated data to ensure that it has been migrated accurately and that the NoSQL database is functioning correctly.
5. Monitor and optimize performance – After the migration is complete, continue to monitor and optimize the performance of the NoSQL database to ensure that it meets the needs of your application.
6. Plan for ongoing maintenance and updates – Migrating to a NoSQL database is not a one-time event – it requires ongoing maintenance and updates to ensure that it continues to meet the needs of your application. Plan for these ongoing efforts as part of your overall migration strategy.

Important Factors While Migrating from SQL to NoSQL:

There are several important factors to consider when migrating from a SQL database to a NoSQL database –

1. Data model – NoSQL databases have different data models than SQL databases, so you will need to determine how to structure your data in the new database.
2. Query language – NoSQL databases often have their own query languages, which may be different from SQL. You will need to learn the new query language and determine how to translate your SQL queries into it.
3. Indexing – SQL databases use indexes to improve the performance of queries, but NoSQL databases often use different indexing techniques. You will need to determine the best way to index your data in the new database.
4. Data integrity – SQL databases typically have stronger data integrity constraints than NoSQL databases. You will need to consider how to ensure data integrity in the new database.
5. Scalability – NoSQL databases are often more scalable than SQL databases. However, you will need to consider how to handle the increased load on the database as it grows.
6. Security – You will need to consider how to secure your data in the new database, including authentication, authorization, and encryption.
7. Cost – NoSQL databases may have different pricing models than SQL databases, so you will need to consider the cost of the new database.
8. Training and support – You may need to provide training for your team on the new database and its query language, and you will need to ensure that there is sufficient support available if you encounter any issues.

Advantages of Migrating from SQL to NoSQL Database:

There are several advantages to migrating from a SQL database to a NoSQL database –

1. Scalability – NoSQL databases are designed to scale horizontally, which means that they can easily handle large amounts of data and high levels of traffic by adding more machines to the database. This makes them well-suited for big data and high-traffic applications.
2. Flexibility – NoSQL databases are generally more flexible than SQL databases because they do not require a fixed schema. This means that you can store data in a variety of formats, including structured, semi-structured, and unstructured data.

3. Performance – NoSQL databases are often faster than SQL databases because they use different indexing and data storage techniques. This can be particularly beneficial for real-time applications that require fast access to data.
4. Simplicity – NoSQL databases are often easier to set up and maintain than SQL databases because they have fewer features and are less complex.
5. Cost – NoSQL databases can be more cost-effective than SQL databases because they are often open-source and have more flexible pricing models.
6. Cloud compatibility – NoSQL databases are well-suited for cloud environments because they can easily scale horizontally and are designed to handle distributed data. This makes them a good choice for cloud-based applications.

DATABASE INDEXES:

▶ Database Indexing: How DBMS Indexing done to improve search query performance? (in-de...

- Indexes are data structures that can increase a database's efficiency in accessing tables.
- It speeds up data retrieval operations by providing quick access to table rows.
- 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.

How to create Index?

In SQL, indexes are created using the `CREATE INDEX` statement.

- Basic Syntax: CREATE INDEX index_name ON table_name (column1, column2, ...);
- Example: CREATE INDEX idx_customer_name ON customers (last_name, first_name);
- Creating a Unique Index: Unique indexes are used for the maintenance of the integrity of the data present in the table as well as for fast performance, it does not allow multiple values to enter into the table. CREATE UNIQUE INDEX idx_unique_customer_email ON customers (email);
- Creating a Composite Index: CREATE INDEX idx_customer_fullname ON customers (last_name, first_name);
- Creating an Index on a Single Column: CREATE INDEX idx_last_name ON customers (last_name);

How to remove, alter, confirm,rename Index?

Remove an index from the data dictionary by using the DROP INDEX command.

- Basic Syntax: DROP INDEX index_name;
- Example: DROP INDEX idx_customer_name;
- Altering an Index: To modify an existing table's index by rebuilding, or reorganizing the index. ALTER INDEX index_name ON table_name REBUILD;
- Confirming Indexes: You can check the different indexes present in a particular table given by the user or the server itself and their uniqueness. SELECT * from USER_INDEXES; It will show you all the indexes present in the server, in which you can locate your own tables too.
- Renaming an Index: You can use the system-stored procedure sp_rename to rename any index in the database.
  EXEC sp_rename index_name, new_index_name, N'INDEX';
  SQL Server Database Syntax: DROP INDEX TableName.IndexName;

How does an Index work?

An index improves the speed of data retrieval operations on a database table. It acts like a lookup table that the database search engine can use to speed up data retrieval.
Here's a step-by-step explanation of how database indexes work:

1. **Index Creation:** The database administrator creates an index on a specific column or set of columns. When an index is created, the database builds a data structure (commonly a B-tree) that stores the indexed column values in a sorted order.
2. **Index Building:** The database management system builds the index by scanning the table and storing the values of the indexed column(s) along with a pointer to the corresponding data.
3. **Query Execution:** When a query is executed, the database engine checks if an index exists for the requested column(s).
4. **Index Search:** If an index exists, the database searches the index for the requested data, using the pointers to quickly locate the data, rather than scanning the entire table.
5. **Data Retrieval:** The database retrieves the requested data, using the pointers from the index.

When Should Indexes be Created?

- A column contains a wide range of values.
- A column does not contain a large number of null values.
- One or more columns are frequently used together in a where clause or a join condition.

When Should Indexes be Avoided?

- The table is small.
- The columns are not often used as a condition in the query.
- The column is updated frequently.

Which data structure do they use?

Internally, an index is usually implemented using a data structure such as a B-tree or a hash table. Here's how these structures work:

- **B-Trees (Balanced Trees):**
  - Structure:
    - B-trees are a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time.
    - Each node contains keys and pointers to child nodes. The keys are sorted within the node.
    - The tree is kept balanced by ensuring that all leaf nodes are at the same depth.
  - Operations:
    - Search: Start at the root and traverse down to the leaf, following the keys.
    - Insert: Add a key to the appropriate leaf node, splitting nodes as necessary to maintain balance.
    - Delete: Remove a key, and if it causes an imbalance, merge or redistribute nodes.
  - Example:
    - Searching for a key involves traversing from the root to the leaf node, where the key or the closest match is found.
- **B+ Trees:**
  - Structure:
    - Similar to B-trees but with all values stored at the leaf nodes, and internal nodes only store keys.
    - Leaf nodes are linked together to allow in-order traversal.

    - Operations:
        - Search: Similar to B-trees, but the search continues to the leaf level where values are stored.
        - Range Queries: Efficiently performed by traversing the linked leaf nodes.
- **Hash Tables:**
    - Structure:
        - Uses a hash function to map keys to specific locations (buckets) in a hash table.
    - Operations:
        - Search: Compute the hash value of the key and directly access the corresponding bucket.
        - Insert: Compute the hash value and place the key in the appropriate bucket.
        - Delete: Compute the hash value and remove the key from the bucket.
    - Example:
    - Directly accessing a key based on its hash value provides constant-time complexity for lookups.
- **Bitmap Indexes:**
    - Structure:
        - Uses bit arrays (bitmaps) to represent the presence or absence of values.
    - Operations:
        - Search: Perform bitwise operations to quickly determine the presence of a value.
        - Range Queries: Combine multiple bitmaps using bitwise operations.
    - Example:
        - Efficient for columns with a limited number of distinct values, as bitmaps can be combined and intersected quickly.

How does an Index Work internally?

In SQL databases, indexes are internally organized in the form of trees. Like actual trees, database indexes have many branch bifurcations, and individual records are represented (or pointed) by the leaves. A database index tree is composed of several nodes connected by pointers.

The index tree is created when the CREATE INDEX statement is executed. The database software has an algorithm that builds the index tree. The first step of the index creation algorithm is to sort the records based on the index key. Next, it builds the structure by creating and populating every node of the tree index. The creation of an index tree can take some time, especially when the table has many records.

If we use an index to search for a specific record in the table, we need to start searching from the root of the index tree. On every branch bifurcation, a decision must be made about which branch to choose. This means logically evaluating the search condition and the range of values in each branch. As a very simple example, imagine you want to find the number 12 in a tree that has two branches. Branch X has numbers from 0-10 and Branch Y has numbers from 11-20. As 12 is greater than 11, you'd search for the number in Branch Y.

1. **Index Creation:**
    a. When an index is created on a column, the database engine constructs the chosen data structure (e.g., B-tree, hash table) to organize the column's data.
    b. The engine scans the table, extracts the indexed column values, and populates the index structure.
2. **Index Maintenance:**
    a. Inserts: When a new row is added to the table, the index structure is updated to include the new value.

   b. Updates: When a row is updated, the index may need to adjust the position of the updated value.ee indexes, nodes may need to be split or merged to maintain balance.
   c. Deletes: When a row is deleted, the corresponding entry in the index is removed.
   d. Rebalancing: For B-tr
3. **Query Execution with Indexes:**
   a. Single-Column Index:
      i. The database engine checks the index to quickly locate the row(s) matching the query.
      ii. Example: SELECT * FROM customers WHERE last_name = 'Smith'; The engine uses the last_name index to find all occurrences of 'Smith'.
   b. Composite Index:
      i. Used for queries involving multiple columns.
      ii. Example: SELECT * FROM customers WHERE last_name = 'Smith' AND first_name = 'John'; The engine uses the composite index on last_name and first_name to locate the rows.
   c. Range Queries:
      i. B-tree indexes support range queries by traversing the tree from the start value to the end value.
      ii. Example: SELECT * FROM orders WHERE order_date BETWEEN '2023-01-01' AND '2023-12-31'; The engine uses the B-tree index on order_date to efficiently fetch the rows within the specified range.
4. **Performance Considerations:**
   a. Selectivity: High selectivity indexes (indexes on columns with many unique values) are more efficient.
   b. Cardinality: Low cardinality indexes (indexes on columns with few unique values) may not be as beneficial.
   c. Maintenance Overhead: Indexes add overhead to insert, update, and delete operations as the index must be kept in sync with the table data.
   d. Storage: Indexes consume additional disk space.

Benefits of using them / Why SQL Indexing is Important?

1. **Improved Query Performance:** Significantly reduces the amount of data SQL Server needs to scan. Increases speed of retrieval operations, making SELECT queries faster.
2. **Efficient Data Retrieval:** Helps in quickly locating the data without scanning the entire table.
3. **Faster Sorting and Grouping:** Indexes help with ORDER BY and GROUP BY operations by reducing the amount of data that needs to be processed.
4. **Facilitates Uniqueness:** Enforces uniqueness on primary keys and unique constraints.
5. **Reduced I/O Cost:** By narrowing down the search to fewer rows, minimizes the number of disk I/O operations required to retrieve data.
6. **Reduced CPU Usage:** By reducing the number of rows that need to be scanned, indexes can decrease CPU usage and optimize resource utilization.

Different types of indexes:

- **Indexes based on Structure and Key Attributes:**
  - **Primary Index:** Automatically created when a primary key constraint is defined on a table. Ensures uniqueness and helps with super-fast lookups using the primary key.
  - **Clustered Index:** Determines the order in which data is physically stored in the table. A clustered index is most useful when we're searching in a range. Only one clustered index can exist per table. CREATE CLUSTERED INDEX idx_primary ON customers (customer_id);

  - **Non-clustered or Secondary Index:** This index does not store data in the order of the index. Instead, it provides a list of virtual pointers or references to the location where the data is actually stored. CREATE INDEX idx_last_name ON customers (last_name);
- **Indexes based on Data Coverage:**
  - **Dense index:** Has an entry for every search key value in the table. Suitable for situations where the data has a small number of distinct search key values or when fast access to individual records is required.
  - **Sparse index:** Has entries only for some of the search key values. Suitable for situations where the data has a large number of distinct search key values.
    - To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for.
    - We start at that record pointed to by the index record, and proceed along with the pointers in the file (that is, sequentially) until we find the desired record.
    - Number of Accesses required=$\log_2(n)+1$, (here n=number of blocks acquired by index file)
- **Specialized Index Types:**
  - **Bitmap Index:** Excellent for columns with low cardinality (few distinct values). Common in data warehousing.
  - **Hash Index:** An index that uses a hash function to map values to specific locations. Great for exact match queries.
  - **Filtered Index:** Indexes a subset of rows based on a specific filter condition. Useful to improve query speed on commonly filtered columns.
  - **Covering Index:** Includes all the columns required by a query in the index itself, eliminating the need to access the underlying table data.
  - **Function-based index:** Indexes that are created based on the result of a function or expression applied to one or more columns of a table.
  - **Single-Column Index:** Indexes on a single column. CREATE INDEX idx_last_name ON customers (last_name);
  - **Composite Index:** Indexes on multiple columns. CREATE INDEX idx_customer_fullname ON customers (last_name, first_name);
  - **Unique Index:** Ensures all values in the index column(s) are unique. CREATE UNIQUE INDEX idx_unique_customer_email ON customers (email);
  - **Full-Text Index:** A index designed for full-text search, allowing for efficient searching of text data. CREATE FULLTEXT INDEX idx_text_search ON articles (content);
  - **Spatial Index:** Used for indexing geographical data types.

How to use them smartly?

To get the most out of database indexes, consider these best practices:

- **Identify Query Patterns:** Analyze the most frequent and critical queries executed against your database to determine which columns to index and which type of index to use.
- **Index Frequently Used Columns:** Consider indexing columns that are frequently used in WHERE, JOIN, and ORDER BY clauses.
- **Index Selective Columns:** Indexes are most effective on columns with a good spread of data values (high cardinality). Indexing a gender column might be less beneficial than one with a unique customer_id.
- **Use Appropriate Index Types:** Choose the right index type for your data and queries.
- **Consider Composite Indexes:** For queries involving multiple columns, consider creating composite indexes that encompass all relevant columns. This reduces the need for multiple single-column indexes and improves query performance.

- **Monitor Index Performance:** Regularly monitor index performance, remove unused indexes and adjust your indexing strategy as the database workload evolves.
- **Avoid Over-Indexing:** Avoid creating too many indexes, as this can lead to increased storage requirements and slower write performance.
    - Indexes take up extra disk space since they're additional data structures that need to be stored alongside your tables.
    - Every time you insert, update, or delete data in a table with an index, the index needs to update too. This can slightly slow down write operations.

To summarize, indexes are a powerful tool to optimize database query performance. But remember to choose the right column and index type, monitor performance, and avoid over-indexing to get the most out of them.

HEARTBEAT:

In distributed systems, a heartbeat is a periodic signal sent at regular intervals between nodes (such as servers, databases, or other computing entities) to indicate that they are alive and functioning properly. Heartbeats are fundamental for maintaining the health and stability of distributed systems.

Why Do We Need Heartbeats?

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

Heartbeats can help with:

- Monitoring: Heartbeat messages help in monitoring the health and status of different parts of a distributed system.
- Detecting Failures: Heartbeats enable a system to identify when a component becomes unresponsive. If a node misses several expected heartbeats, it's a sign that something might be wrong.
- Triggering Recovery Actions: Heartbeats allow the system to take corrective actions. This could mean moving tasks to a healthy node, restarting a failed component, or letting a system administrator know that they need to step in.
- Load Balancing: By monitoring the heartbeats of different nodes, a load balancer can distribute tasks more effectively across the network based on the responsiveness and health of each node.

How Does a Heartbeat Work?

- **Generation and Transmission:**
    - Each node in the system periodically generates and sends a heartbeat message to a designated recipient (another node, a central server, or a monitoring system).
    - The message usually includes identification information, timestamp, and sometimes the node’s status (e.g., resource utilization).
- **Reception and Monitoring:**
    - The recipient node or system receives the heartbeat messages and logs them.
    - It monitors the timing and content of these messages to assess the health of the sender nodes.
- **Timeout Detection:**
    - If the recipient does not receive a heartbeat within a predefined timeout period, it assumes that the sender node has failed or is unreachable.
    - This can trigger various actions, such as sending alerts, initiating failover processes, or marking the node as down.

Internal Mechanism:

- **Configuration:**
    - Nodes are configured with heartbeat intervals and timeout periods.
    - Example configuration parameters might include heartbeat_interval (e.g., every 5 seconds) and timeout_period (e.g., 15 seconds).
- **Protocol:**
    - Heartbeat messages follow a protocol to ensure they are recognized and processed correctly.
    - Protocols may include TCP, UDP, or custom application-layer protocols.
- **Failover and Recovery:**
    - When a node fails to send heartbeats, the system can redistribute tasks, reallocate resources, or initiate recovery processes to maintain overall system stability.

While conceptually simple, heartbeat implementation has a few nuances:

- **Frequency:** How often should heartbeats be sent? There needs to be a balance. If they're sent too often, they'll use up too much network resources. If they're sent too infrequently, it might take longer to detect problems.
- **Timeout:** How long should a node wait before it considers another node 'dead'? This depends on expected network latency and application needs. If it's too quick, it might mistake a live node for a dead one, and if it's too slow, it might take longer to recover from problems.
- **Payload:** Heartbeats usually just contain a little bit of information like a timestamp or sequence number. But, they can also carry additional data like how much load a node is currently handling, health metrics, or version information.

Types of Heartbeats:

- **Push heartbeats:** Nodes actively send heartbeat signals to the monitor.
- **Pull heartbeats:** The monitor periodically queries nodes for their status.
- **Unidirectional Heartbeats:**
    - Sent from one node to another or to a central server.
    - Simple and easy to implement.
    - Example: A client sending a heartbeat to a server.
- **Bidirectional Heartbeats:**
    - Nodes exchange heartbeat messages with each other.
    - Provides mutual health checks and can enhance fault tolerance.
    - Example: Peer nodes in a distributed database cluster.
- **Multicast/Broadcast Heartbeats:**
    - Heartbeats sent to multiple nodes simultaneously using multicast or broadcast methods.
    - Efficient for large-scale systems but may introduce network overhead.
    - Example: Nodes in a distributed ledger system.

Heartbeat Protocols:

In distributed systems, heartbeat protocols are used as a means of communication to transfer heartbeat messages amongst nodes or components. These protocols make it easier for distributed system entities to coordinate, detect failures, and monitor system health. Several distributed systems frequently employ one of the following heartbeat protocols:

1. **Simple Heartbeat Protocol (SHP):**
    a. For the purpose of transmitting heartbeat signals between nodes in a distributed system, the Simple Heartbeat Protocol is a straightforward and lightweight protocol.
    b. Typically, SHP uses a straightforward message exchange to report the availability and liveness of nodes at regular intervals.

   c. This protocol is simple to use and appropriate in situations where a simple heartbeat mechanism is sufficient.

2. **Ping/Echo Protocol:**
   a. Sending a “ping” message from one node to another and waiting for an “echo” response from the receiving node is the Ping/Echo protocol, also called the Ping-Pong protocol.
   b. For network-level communication, this protocol is commonly implemented using the Internet Control Message Protocol (ICMP), and for inter-process communication, it is typically implemented using custom application-layer protocols.
   c. In networked environments, the Ping/Echo protocol is frequently used for basic connectivity checks and health monitoring.
3. **UDP-based Heartbeat Protocol:**
   a. User Datagram Protocol (UDP) is used by UDP-based heartbeat protocols to facilitate communication between nodes.
   b. These protocols usually entail periodic transmission of lightweight UDP packets with heartbeat messages inside of them.
   c. Protocols for UDP-based heartbeats are appropriate in situations where low latency and low overhead are required.
4. **TCP-based Heartbeat Protocol:**
   a. Transmission Control Protocol (TCP) is used by TCP-based heartbeat protocols to enable communication between nodes.
   b. In these protocols, nodes create a TCP connection and communicate by sending each other heartbeat messages over the connection.
   c. TCP-based heartbeat protocols are appropriate in situations where dependability is crucial because they guarantee message delivery and provide dependable communication.
5. **Raft Protocol:**
   a. A consensus protocol called Raft is used in distributed systems to accomplish replication and fault tolerance.
   b. Heartbeat messages are used by the Raft protocol in the leader election and replication procedures.
   c. In a distributed system based on Raft, nodes communicate via heartbeat messages to track the health of the leader and identify any malfunctions.
6. **Apache ZooKeeper Heartbeats:**
   a. Heartbeat messages are used by Apache ZooKeeper, a distributed coordination service, for session management and leader election.
   b. Clients of ZooKeeper send heartbeat messages on a regular basis to keep their session with the ZooKeeper ensemble going.
   c. ZooKeeper servers also use heartbeat messages to elect a leader and check if other servers are still active.

CIRCUIT BREAKER:

What is Circuit Breaker Pattern in Microservices? - GeeksforGeeks

In microservices architecture, a circuit breaker is a design pattern used to handle faults that may occur when calling remote services. The purpose of the circuit breaker is to prevent a cascade of failures in a distributed system by providing a fallback mechanism when a service is unavailable or experiencing issues.

The Circuit Breaker pattern in microservices is a fault-tolerance mechanism that monitors and controls interactions between services. It dynamically manages service availability by temporarily interrupting requests to failing services, preventing system overload, and ensuring graceful degradation in distributed environments.

Characteristics of Circuit Breaker Pattern:

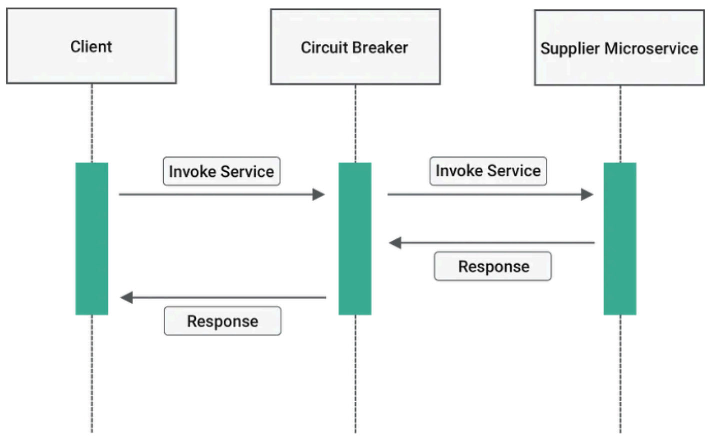
- **Fault Tolerance:** Enhances fault tolerance by isolating and managing failures in individual services.
- **Monitoring:** Continuously monitors interactions between services to detect issues in real time.
- **Failure Isolation:** Temporarily stops requests to failing services, preventing cascading failures and minimizing disruptions.
- **Fallback Mechanism:** Provides fallback responses or error messages to clients during service failures, ensuring graceful degradation.
- **Automatic Recovery:** Automatically transitions back to normal operation when the failing service recovers, improving system reliability.

Working and Different States in Circuit Breaker Pattern:

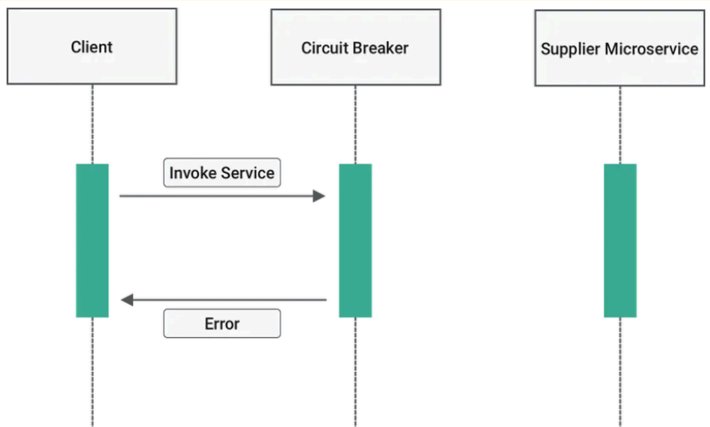
With the help of this pattern, the client will invoke a remote service through a proxy. This proxy will basically behave as an electrical circuit breaker. So, when the number of failures crosses the threshold number, the circuit breaker trips for a particular time period. Then, all the attempts to invoke the remote service will fail within this timeout period. After the timeout expires, the circuit breaker allows a limited number of test requests to pass through it. If those requests succeed, the circuit breaker resumes back to the normal operation. Otherwise, if there is a failure, the timeout period begins again.

The Circuit Breaker pattern typically operates in three main states: Closed, Open, and Half-Open. Each state represents a different phase in the management of interactions between services.

- **Closed State:**



  - In the Closed state, the circuit breaker operates normally, allowing requests to flow through between services.
  - During this phase, the circuit breaker monitors the health of the downstream service by collecting and analyzing metrics such as response times, error rates, or timeouts.
  - If the monitored metrics remain within acceptable thresholds, indicating that the downstream service is healthy, the circuit breaker stays in the Closed state and continues to forward requests.
- **Open State:**
  - When the monitored metrics breach predetermined thresholds, signaling potential issues with the downstream service, the circuit breaker transitions to the Open state.

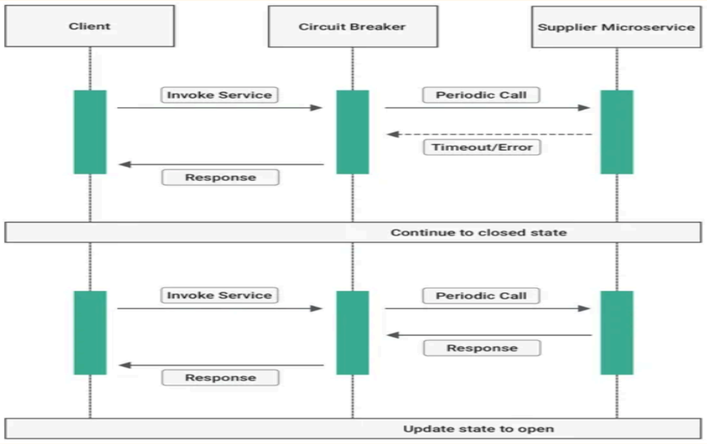


  - In the Open state, the circuit breaker immediately stops forwarding requests to the failing service, effectively isolating it.
  - Instead of allowing requests to reach the failing service and potentially exacerbate the issue, the circuit breaker

provides a predefined fallback response or an error message to the caller.

  - This helps prevent cascading failures and maintains system stability by ensuring that clients receive timely feedback, even when services encounter issues.

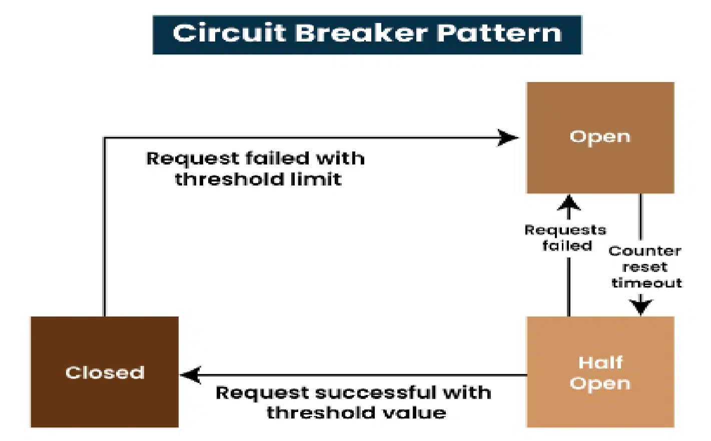
- **Half-Open State:**
  - After a specified timeout period in the Open state, transitions to Half-Open state.



  - Allows a limited number of trial requests to pass through to the downstream service.
  - Monitors responses to determine service recovery.
  - If trial requests succeed, indicating service recovery, transitions back to Closed state.
  - If trial requests fail, service issues persist, may transition back to Open state or remain in Half-Open state for further evaluation.

State Transition:

1. **Closed State to Open State:**
   a. Transition Trigger: Request failed with threshold limit.
   b. Explanation: When the monitored metrics breach the predefined thresholds while the Circuit Breaker is in the Closed state, indicating potential issues with the downstream service, it transitions to the Open state. This means that the Circuit Breaker stops forwarding requests to the failing service(that is experiencing issues or failures)and provides fallback responses to callers.



2. **Half-Open State to Closed State:**
   i. Transition Trigger: Request successful with threshold value.
   ii. Explanation: After a specified timeout period in the Open state, the Circuit Breaker transitions to the Half-Open state. In the Half-Open state, a limited number of trial requests are allowed to pass through to the downstream service. If these trial requests are successful and the service appears to have recovered, the Circuit Breaker transitions back to the Closed state, allowing normal traffic to resume.
3. **Half-Open State to Open State:**
   a. Transition Trigger: Request failed.
   b. Explanation: While in the Half-Open state, if the trial requests to the downstream service fail, indicating that the service is still experiencing issues, the Circuit Breaker transitions back to the Open state. This means that requests will again be blocked, and fallback responses will be provided until the service's health improves.
4. **Open State to Half-Open State:**
   a. Transition Trigger: Counter reset timeout.
   b. Explanation: After a specified timeout period in the Open state, the Circuit Breaker transitions to the Half-Open state. This timeout period allows the Circuit Breaker to

periodically reevaluate the health of the downstream service and determine if it has recovered.

Steps to Implement Circuit Breaker Pattern:

1. **Identify Dependencies:** Going for the external services that will bring interactions and make the microservice functional in turn.
2. **Choose a Circuit Breaker Library:** Choose a circuit breaker library from these existing libraries, or create your own framework that you are familiar with, based on your programming language and platform. Some of the options that rank high for Java include Hystrix, resilience4j is a favorite for JVM-based languages, Polly is the preferred choice for .NET, and many others.
3. **Integrate Circuit Breaker into Code:** Make sure to insert the selected circuit breaker library into your microservices code base. Circuit breaking in this case is usually done by re-configuring or adding annotations to your programming structure such as service methods or HTTP clients.
4. **Define Failure Thresholds:** Set boundaries for faults and time-outs that turn the mechanism of the circuit breaker to open. This also can be a measure of setting the number of the failed constructs consecutively or the responsive time for external calls.
5. **Implement Fallback Mechanisms:** Include whenever the circuit has open or close requests, the fallback mechanism should be implemented. Some alternatives could be to send back cached data, to give responses that can be chosen later, or to forward requests to several services at the same time.
6. **Monitor Circuit Breaker Metrics:** Use the statistics built into the circuit breaker library to see the health and behavior of your services. Such evaluation measurements encompass metrics, for instance, number of successful/unsuccessful requests, status of the circuit breaker, and error rates.
7. **Tune Configuration Parameters:** Tuning configuration parameters like timeouts, thresholds, and retry methods in accordance to behavior of your microservices and your application requirements.
8. **Test Circuit Breaker Behavior:** Perform live testing of your circuit breaker during different operating states including normal function, failure scenarios (high load), and fault condition. Develop a circuit breaker that adequately addresses cases of failures while smoothly recovering when the service concerned becomes available.
9. **Deploy and Monitor:** Move/deploy your microservice with circuit breaker, into your production environment. Relentlessly conduct a monitoring of services' operation and behavior; make timely tweaks of parameters considered the most suitable to guarantee high resilience and stability.

When to use Circuit Breaker Pattern:

The Circuit Breaker pattern is particularly useful in microservices architectures in the following scenarios: The Circuit Breaker pattern is particularly useful in microservices architectures in the following scenarios:

- **Remote Service Calls:** While microservices set out to communicate with other services via the network, they encounter failures such as network downs, service unveil, or slow response. This is where the Circuit Breaker pattern helps to customize the solutions. For instance, it is able to contain a given failure, by making a provision for services to regain power and presence flawlessly.
- **High Availability Requirements:** The Circuit Breaker design pattern, which is applied in system configurations that have high service availability, helps around with failures. This gives the system the ability to self-heal and still carry on even when the individual services have crashed and burned.

- **Scalability and Load Handling:** As the microservices gateways may be experiencing loading or other situations, circuit breakers may be used to control traffic when microservices architecture is used. With a redirect of inbound requests when services become saturated, circuit breaker severs traffic stalemate and safeguards system integrity.
- **Fault Isolation:** The circuit breaker pattern is beneficial for error removal and failures that may occur in microservices architectures as it isolates faults. That ensures that failures in the parts not affecting other operations do not propagate to other areas of the system, which improves system resilience and fault tolerance.
- **Asynchronous Processing:** Use of circuit breakers in such asynchronous systems that perform messaging queuing or event processing can be a very handy tool in handling failures in message brokers or event queues. It is precisely this fact which guarantees the system that it is still processing messages even though there might be hiccups in the system.

IDEMPOTENCY:

Idempotency is a property of certain operations or API requests, which guarantees that performing the operation multiple times will yield the same result as if it was executed only once. This principle is especially vital in Distributed Systems and APIs, as it helps maintain consistency and predictability in situations such as network issues, request retries, or duplicated requests.This principle simplifies error handling, concurrency management, debugging, and monitoring, while also enhancing the overall user experience. In REST APIs, HTTP methods like POST and PATCH are NOT idempotent, GET, PUT, DELETE, HEAD, OPTIONS and TRACE are idempotent.

Why is Idempotency Important?

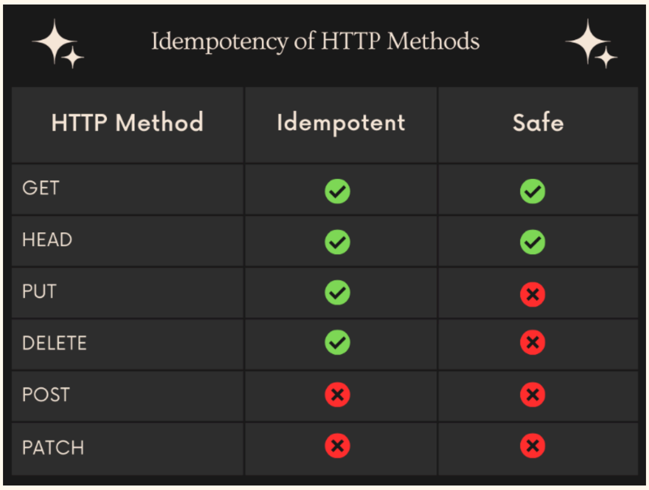
A resource may be called multiple times if the network is interrupted. In this scenario, non-idempotent operations can cause significant unintended side-effects by creating additional resources or changing them unexpectedly. When a business relies on the accuracy of its data, non-idempotency posts a significant risk.

Idempotency is crucial for ensuring the following:

- **Consistency:** The system maintains a predictable state, even when faced with request duplication or retries.
- **Error Handling:** Idempotent operations simplify error handling, as clients can safely retry requests without the risk of unintended side effects.
- **Fault Tolerance:** Idempotent APIs can better cope with network issues and other disruptions, ensuring a more reliable user experience.

Idempotent Methods in REST:

In REST APIs, HTTP methods like GET, HEAD, PUT, and DELETE are inherently idempotent.



Idempotency of HTTP Methods

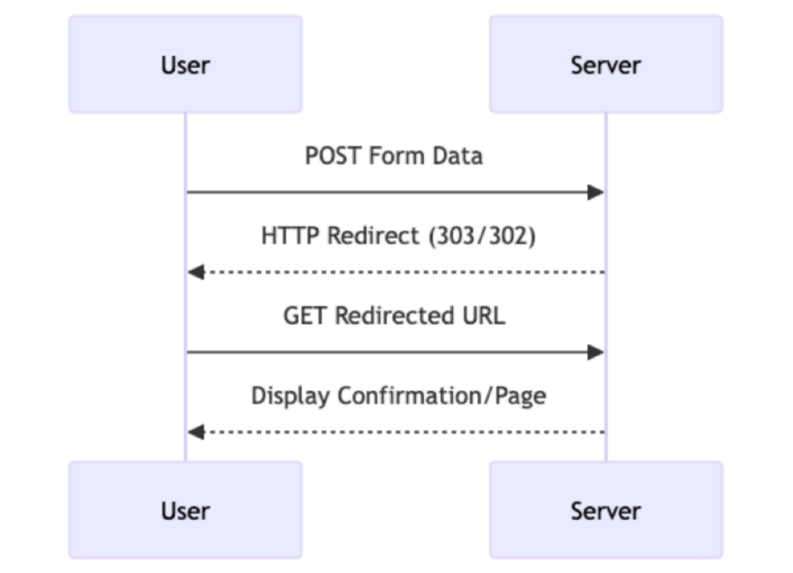
HTTP Method	Idempotent	Safe
GET	✓	✓
HEAD	✓	✓
PUT	✓	✗
DELETE	✓	✗
POST	✗	✗
PATCH	✗	✗

- **GET:** Used to retrieve data from a server. Multiple GET requests to the same resource are safe and should return the same data, assuming no changes have been made to the resource in the meantime.
- **HEAD:** Similar to GET, but it retrieves only the header information about a resource. Since it does not return a body, it's inherently safe and idempotent.
- **PUT:** Replaces a resource's current representation with the request payload. Repeatedly putting the same data to the same resource endpoint will leave the resource in the same state.

- **DELETE:** Removes a resource. Deleting the same resource multiple times results in the same outcome: the resource is removed after the first successful request, and subsequent DELETE requests typically return a 404 Not Found or 204 No Content status, indicating that there's no resource to delete.
    - Delete without the resource identifier: Some systems may have DELETE APIs like this: DELETE /item/last. In this case, calling operation N times will delete N resources – hence DELETE is not idempotent in this case. In this case, a good suggestion might be to change the above API to POST – because POST is not idempotent: POST /item/last

How to Make a POST Operation Idempotent?

A POST operation is not inherently idempotent, since it is typically used to create resources. But some implementations of POST can be designed to be idempotent.Post/Redirect/Get pattern, also referred to as the PRG pattern, is particularly useful for handling form submissions and can mitigate issues caused by users refreshing or bookmarking pages that make changes to the server's state.



Sequence diagram showing how the PRG pattern works to "convert" a POST operation to an idempotent GET operation

Post: When the user submits a form to place an order, the browser sends a POST request to the server.

Redirect: After the server processes the POST request (for example, placing the order), it sends a redirect response to the browser, usually with a 303/302 HTTP status code, directing it to a new URL. This URL is typically an order confirmation page.

Get: The browser then makes a GET request to the URL provided by the redirect. The user sees the page that confirms their order or brings them back to a safe state where no duplicate orders can be accidentally created.

The key benefit of using the PRG pattern is that it turns the POST request into a GET request, which is idempotent. This means that refreshing the page at the end of the process will not cause the same order to be submitted more than once, because refreshing will only repeat the GET request, not the initial POST request that submitted the order.

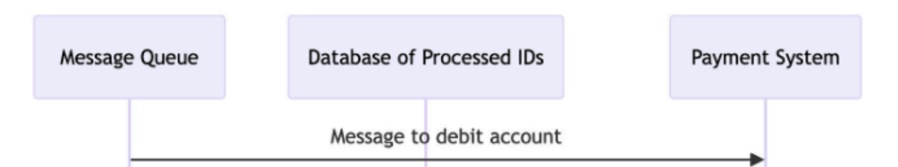
This pattern enhances the user experience by preventing common mistakes, such as double-clicking a submission button or refreshing the page, creating unwanted duplicate orders.

It also makes the application more robust and user-friendly, as users can safely refresh the confirmation page or bookmark it without worrying about the order being duplicated.

Message Queueing Systems:

A message queue can contribute to making a system idempotent by ensuring that even if a message (representing a request) is delivered multiple times, the operation it triggers is executed only once, or its effect is the same regardless of how many times it's executed.
This is crucial in distributed systems where network failures, system crashes, or other issues can lead to the same message being processed multiple times.

1. The message queue sends a message to the payment system to debit an account.



2. This message has a unique Transaction ID. The Database of Processed IDs maintains a record of Transaction IDs that have been processed. The Payment System checks the Transaction ID against the Database of Processed IDs and checks if this payment has already been processed.
3. If the message has a transaction ID in the Processed ID Database, it is ignored and treated as already processed. The payment has already been made and does not need to be repeated. The Payment System sends an acknowledgement (ACK) back to the queue to inform it that the message has been ignored. The message queue needs to know that the message has been handled before it deletes the message on its side.
4. If the message does not have a Transaction ID in the Processed ID database, this means the payment has not been processed before. The payment is therefore processed and the transaction ID is added to the Database of Processed IDs. Ideally, these two steps should be done in a single atomic transaction. This prevents an unwanted state where the payment is processed but the Transaction ID is never added to the Database of Processed IDs because of a database failure, networking issue or any other fault.
5. In the final step, the Payment System sends an acknowledgement (ACK) back to the queue to inform it that the message has been successfully processed. This acknowledgement informs the queue that the message has been successfully received, processed, and no longer needs to be kept in the queue for future delivery. This prevents the message from being sent again, ensuring that the system is idempotent.

In this example, the system ensures idempotency by:

- Checking Transaction IDs for new payments against a database of payments already made
- Sending the acknowledgement to the queue after the message is ignored or processed by the Payment System. Thus ensures that the same message is only sent to the Payment System once.

Can you provide an example of idempotency in practice?

Consider an API for monetary transactions. If a user requests to transfer $100 from Account A to Account B, and due to network issues, the request is duplicated, idempotency ensures that only one $100 transfer occurs, preventing the unintended transfer of $200.

What benefits does idempotency offer?

Idempotency simplifies error handling, concurrency management, debugging, and monitoring in operations and APIs. It enhances user experience by maintaining consistency and predictability, even in the face of network disruptions and request retries.

What’s the difference between idempotency and safety?

Idempotency ensures that repeated operations yield the same outcome. Safety, on the other hand, refers to operations that don’t modify the returned value. While all safe methods are idempotent, not all idempotent methods are safe.

Which HTTP methods are idempotent?

HTTP methods such as GET, HEAD, PUT, DELETE, OPTIONS, and TRACE are considered idempotent. They can be retried or executed multiple times without causing unintended side effects.

Is POST an idempotent method?

No, POST is not inherently idempotent. Making the same POST request multiple times can result in different outcomes or create multiple instances of submitted data.

DATABASE SCALING:

Database Scaling is the process of adding or removing from a database's pool of resources to support changing demand. A database can be scaled up or down to accommodate the needs of the application that it's supporting.

What is a Scalable Database?

A scalable database is a type of database system designed to handle growing amounts of data and increasing numbers of users or requests without sacrificing performance or reliability. Scalability in databases typically refers to two aspects: vertical scalability and horizontal scalability.

- **Vertical Scalability:** Involves adding more resources (CPU, RAM, storage) to a single server to handle increased load. While vertical scaling can provide immediate relief, it has inherent limits and can become prohibitively expensive.
- **Horizontal Scalability:** Involves distributing the workload across multiple servers or nodes. Horizontal scaling offers greater flexibility and can accommodate virtually unlimited growth by adding more servers to the cluster.

How to Design a Scalable Database?

Designing a scalable database involves careful consideration of various factors, including Data Partitioning, architecture choices, and scalability strategies. Here's a step-by-step guide to design a scalable database:

- **Data Partitioning/Sharding:**
    - Data partitioning, also known as sharding, involves breaking down the dataset into smaller subsets called shards and distributing them across multiple servers or nodes.
    - Sharding enables parallel queries and reduces contention because different shards can be queried simultaneously without interfering with each other.
    - Let's consider an e-commerce platform where product data is partitioned based on categories. Each shard contains product information for a specific category, enabling parallel queries and reducing contention.
- **Replication:**
    - Replication involves creating redundant copies of data across multiple servers to ensure fault tolerance and high availability.
    - In a replicated database setup, write operations are typically directed to a primary node, while read operations can be distributed across replica nodes, improving read scalability and fault tolerance.
    - Replication can be synchronous or asynchronous, depending on the consistency requirements of the application.
- **Caching:**
    - Caching involves storing frequently accessed data in memory to reduce latency and improve performance.
    - Caching can be implemented using in-memory data stores like Redis or Memcached, which offer fast read access and can significantly reduce the database workload by serving frequently accessed data directly from memory.
    - For example, a social media platform may cache user profiles or feed data to accelerate retrieval and minimize database load.
- **Load Balancing:**
    - Load balancing involves distributing incoming traffic across multiple database servers to prevent overload on any single server.
    - A load balancer sits between clients and database servers, intelligently routing requests based on server health, capacity, and other factors.
    - Load balancing helps scale horizontally by adding more servers to the database cluster and distributing the workload evenly across them.

    - Modern load balancers often support dynamic scaling and auto-scaling features, allowing the infrastructure to adapt to changing traffic patterns and maintain optimal performance levels.

DATA REPLICATION:

Data replication, also known as database replication, is a process of duplicating data across multiple servers or locations. This ensures that copies of the data are available in multiple locations, which can improve data availability, fault tolerance, and performance.

Types of Data Replication:

- **Synchronous Replication:**
    - Changes to data are made simultaneously across all replicas.
    - Ensures strong consistency but can introduce latency.
    - Example: Financial systems where consistency is crucial.
- **Asynchronous Replication:**
    - Changes to data are propagated to replicas after the primary data source is updated.
    - Reduces latency and improves performance but may lead to temporary inconsistencies.
    - Example: Content delivery networks (CDNs) where eventual consistency is acceptable.
- **Semi-Synchronous Replication:**
    - A hybrid approach where the primary node waits for acknowledgment from at least one replica before committing a transaction.
    - Balances consistency and performance.
- **Transactional Replication:**
    - In Transactional replication users receive full initial copies of the database and then receive updates as data changes.
    - Data is copied in real-time from the publisher to the receiving database(subscriber) in the same order as they occur with the publisher therefore in this type of replication, transactional consistency is guaranteed.
    - Transactional replication is typically used in server-to-server environments. It does not simply copy the data changes, but rather consistently and accurately replicates each change.
- **Snapshot Replication**:
    - Snapshot replication distributes data exactly as it appears at a specific moment in time and does not monitor for updates to the data. The entire snapshot is generated and sent to Users.
    - Snapshot replication is generally used when data changes are infrequent. It is a bit slower than transactional because on each attempt it moves multiple records from one end to the other end.
    - Snapshot replication is a good way to perform initial synchronization between the publisher and the subscriber.
- **Merge Replication:**
    - Data from two or more databases is combined into a single database.
    - Merge replication is the most complex type of replication because it allows both publisher and subscriber to independently make changes to the database.
    - Merge replication is typically used in server-to-client environments. It allows changes to be sent from one publisher to multiple subscribers.
- **Bidirectional Replication:**
    - Data can be modified at any node, and changes are synchronized across all nodes.

    - Ensures high availability and fault tolerance but requires conflict resolution mechanisms to handle concurrent updates.
    - Example: Multi-master database systems.
- **Unidirectional Replication:**
    - Data is replicated from a primary node to one or more secondary nodes.
    - Secondary nodes are typically read-only, ensuring data consistency without the complexity of conflict resolution.
    - Example: Read replicas in a web application to distribute read traffic.

How Data Replication Works?

- **Replication Process:**
    - Primary Node: The main node where the original data is stored.
    - Replica Nodes: Nodes where the data is copied to.
    - Synchronization: Ensuring that data changes in the primary node are propagated to replica nodes. This can be done synchronously (immediate update) or asynchronously (update after some delay).
- **Replication Techniques:**
    - Full Replication: Entire database or dataset is copied to multiple nodes.
    - Partial Replication: Only a subset of the data is replicated to different nodes, often used to reduce storage requirements and improve performance.

Replication Schemes:

Full Replication: The most extreme case is replication of the whole database at every site in the distributed system. This will improve the availability of the system because the system can continue to operate as long as at least one site is up.

- Advantages:
    - High Availability of Data.
    - Improves the performance for retrieval of global queries as the result can be obtained locally from any of the local sites.
    - Faster execution of Queries.
- Disadvantages:
    - Concurrency is difficult to achieve in full replication.
    - Slow update process as a single update must be performed at different databases to keep the copies consistent.
    - The data can be easily recovered.
    - Concurrency can be achieved in no replication.
    - Since multiple users are accessing the same server, it may slow down the execution of queries.
    - The data is not easily available as there is no replication.
    - The number of copies of the fragment depends upon the importance of data.
    - To provide a consistent copy of data across all the database nodes.
    - To increase the availability of data.
    - The reliability of data is increased through data replication.
    - Data Replication supports multiple users and gives high performance.
    - To remove any data redundancy,the databases are merged and slave databases are updated with outdated or incomplete data.
    - Since replicas are created there are chances that the data is found itself where the transaction is executing which reduces the data movement.
    - To perform faster execution of queries.
    - More storage space is needed as storing the replicas of same data at different sites consumes more space.

    - Data Replication becomes expensive when the replicas at all different sites need to be updated.
    - Maintaining Data consistency at all different sites involves complex measures.

No replication: No replication means, each fragment is stored exactly at one site.

- Advantages:
    - Concurrency has been minimized as only one site to be updated
    - Only one site hence easy to recover data.
- Disadvantages:
    - Poor availability of data as a centralized server only has data.
    - Slow down query execution as multiple clients accessing the same server.

Partial replication: Partial replication means some fragments are replicated whereas others are not. Only a subset of the database is replicated at each site. This reduces storage costs but requires careful planning to ensure data consistency.

- Advantages of partial replication:
    - Number of replicas created for a fragment directly depends upon the importance of data in that fragment.
    - Optimized architecture gives advantages of both full replication and no replication scheme.

Replication Mechanisms:

- **Master-Slave Replication:**
    - One master server where all writes occur, and multiple slave servers that replicate the data.
    - Slaves handle read operations, reducing the load on the master.
- **Multi-Master Replication:**
    - Allows writes to be performed on any node, and changes are propagated to all other nodes.
    - Provides high availability and load distribution but requires conflict resolution mechanisms.
- **Peer-to-Peer Replication:**
    - Every node in the system can read and write data, and replication occurs among all peers.
    - Often used in distributed database systems for high availability and fault tolerance.

DATA REDUNDANCY:

Data redundancy refers to the duplication of data within a database or across multiple storage systems. While it can introduce extra storage costs and potential data management complexities, it plays a crucial role in enhancing the reliability, availability, and performance of information systems.

How Does Data Redundancy Occur?

Data redundancy occurs when the same piece of data is stored in multiple places within a database or across different systems. Here are some common scenarios where data redundancy can occur:

- **Database Design:**
    - Denormalization: To optimize read performance, databases may store redundant data in multiple tables.
    - Lack of Normalization: Poor database design that does not adhere to normalization rules can lead to unintentional redundancy.
- **Backup and Replication:**

    - Backup Copies: Regular backups create redundant copies of the data for recovery purposes.
    - Replication: Data is copied to multiple servers or locations to ensure availability and fault tolerance.
- **Distributed Systems:**
    - Caching: Systems may cache data in multiple places to improve access speed, resulting in redundancy.
    - Redundant Storage: Systems like RAID or distributed databases store multiple copies of data across different nodes.
- **Application-Level Redundancy:**
    - Data Duplication: Different applications or services may store the same data independently, leading to redundancy.

Following are a few examples of how data redundancy occurs -

- Forms that collect the same information in different fields (e.g., first name/last name, first/last)
- Multiple backups of the same data by individuals or groups who are unaware that the other is creating a backup
- Older versions of backups being saved rather than deleted or overwritten by newer versions
- Poor coding within a data management system that causes data not to update correctly, resulting in discrepancies within the database
- Separate systems that collect and store the same information (e.g., customer information collected and stored in finance, sales, and marketing systems)

Why Data Redundancy Can Be a Problem?

- **Increased Storage Costs:** Storing multiple copies of data consumes more storage resources, leading to higher costs.
- **Data Inconsistency:** Redundant data can lead to inconsistencies if not properly synchronized, resulting in conflicting information across the system.
- **Complex Data Management:** Managing multiple copies of data requires additional processes and resources to ensure synchronization and integrity.
- **Performance Issues:** Redundant data can increase the complexity of data management tasks, potentially affecting system performance.
- **Data Anomalies:** Redundancy can lead to anomalies such as update, insertion, and deletion anomalies, complicating database operations.

How Data Redundancy Is Resolved for a Database?

- **Normalization:**
    - Process: Organize the database into tables and columns to minimize redundancy.
    - Normal Forms: Apply normal forms (1NF, 2NF, 3NF, etc.) to structure the data logically.
- **Use of Foreign Keys:**
    - Process: Establish relationships between tables using foreign keys to ensure data integrity without redundancy.
- **Constraints and Triggers:**
    - Constraints: Use unique constraints to prevent duplicate entries.
    - Triggers: Implement triggers to automatically synchronize redundant data, if necessary.
- **Data Deduplication Tools:**
    - Use tools and scripts to identify and remove redundant data periodically.

How Data Redundancy Is Resolved Between Different Systems?

- **Data Integration:**
  - ETL Processes: Use Extract, Transform, Load (ETL) processes to consolidate data from multiple systems into a single, integrated data warehouse.
  - Data Warehousing: Store integrated data centrally to reduce redundancy and improve data consistency.
- **Data Synchronization:**
  - Real-Time Synchronization: Use middleware or data synchronization services to keep data consistent across systems in real-time.
  - Batch Synchronization: Schedule regular batch processes to synchronize data between systems.
- **API Integration:**
  - Use of APIs: Implement APIs to allow systems to communicate and share data in real-time, reducing the need for redundant data storage.
  - Microservices Architecture: Adopt a microservices architecture where services communicate via APIs, each handling specific data and reducing redundancy.
- **Master Data Management (MDM):**
  - Centralized Master Data: Establish a single source of truth for critical data entities (e.g., customer, product) and ensure all systems refer to this master data.
  - Data Governance: Implement data governance policies to manage the creation, updating, and deletion of master data consistently.
- **Data Deduplication:**
  - Deduplication Tools: Use specialized data deduplication tools to identify and remove duplicate data across different systems.
  - Example: Implement deduplication algorithms that compare data across systems to identify and consolidate duplicates.

Aspect	Data Redundancy	Data Replication
Purpose	Ensures system continuity by providing backup components	Enhances performance, fault tolerance, and accessibility
Concept	Involves extra copies or resources for backup	Involves creating exact duplicates of data or processes
Focus	Primarily on system reliability and fault tolerance	Primarily on system efficiency and performance
Example	Backup power supplies, redundant hard drives	Mirrored databases, load-balanced servers
Implementation	Duplicate hardware components, backup systems	Data mirroring, distributed computing
Benefit	Minimizes downtime and data loss during failures	Optimizes performance, improves fault tolerance

Impact on Scalability and Performance:

1. Scalability:
    a. Impact of Redundancy: Redundancy can facilitate scalability by providing additional resources that can be activated when needed, allowing systems to handle increased loads or demands without compromising performance. For instance, having redundant servers or network paths enables systems to scale horizontally by adding more servers or network connections as the workload increases.
    b. Impact of Replication: Replication supports scalability by distributing workload across multiple copies of data or resources, reducing the burden on individual components and enabling systems to handle more concurrent requests or users. For example, database replication allows read operations to be distributed across multiple replicas, improving scalability for read-heavy workloads.
2. Performance:
    a. Impact of Redundancy: Redundancy can have a mixed impact on performance. While it ensures high availability and fault tolerance, it may introduce some overhead due to the need to synchronize data or resources across redundant components. However, well-designed redundancy mechanisms, such as load balancing and failover, can mitigate performance degradation by efficiently distributing workload and seamlessly transitioning between redundant components.
    b. Impact of Replication: Replication generally improves performance by reducing latency and improving data accessibility. By maintaining multiple copies of data or resources, replication enables systems to serve requests from the nearest or least loaded replica, minimizing response times. Additionally, replication can enhance performance by allowing parallel processing of requests across multiple replicas, especially in distributed computing environments.

Role in Fault Tolerance:

1. Redundancy:
    a. Component Redundancy: Redundancy involves having backup components or systems that can take over if primary components fail. For example, redundant power supplies or servers can keep a system running if one fails.
    b. Fault Isolation: Redundancy helps isolate faults by ensuring that if one component fails, it does not bring down the entire system. Failures are contained to the affected component, allowing the rest of the system to continue functioning.
    c. Automatic Failover: Redundancy mechanisms often include automatic failover processes that detect failures and switch to backup components seamlessly. This provides continuous operation and reduces downtime.
2. Replication:
    a. Data Redundancy: Replication involves creating duplicate copies of data or resources across multiple locations. If one copy becomes unavailable due to failure or errors, other copies can still be accessed, ensuring data availability.
    b. Load Balancing: Replication can distribute workload across multiple replicas, preventing overload on any single component. This helps in avoiding performance degradation or failures caused by excessive load on a single resource.
    c. Geographic Redundancy: Replicating data or resources across geographically distributed locations provides protection against regional failures, such as natural disasters or network outages. Users can access replicated resources from alternate locations if one region becomes inaccessible.

SHARDING: What is Sharding? - Database Sharding Explained - AWS (amazon.com)

Database sharding is a horizontal scaling technique used to split a large database into smaller, independent pieces called shards. This method improves performance and scalability by distributing the load and storage across multiple nodes.
Sharding is widely used by large-scale applications and services that handle massive amounts of data, such as:

- Social Networks: Instagram uses sharding to manage billions of user profiles and interactions.
- E-commerce Platforms: Amazon employs sharding to handle massive product catalogs and customer data.
- Search Engines: Google relies on sharding to index and retrieve billions of web pages.
- Gaming: Online gaming platforms use sharding to handle millions of players and vast amounts of game data.

How does Sharding Work?

Here's an in-depth look at how sharding works:

1. Concept of Sharding:
    a. Shard: A horizontal partition of data in a database. Each shard is held on a separate database server instance to spread load.
    b. Sharding Key: A unique identifier used to determine which shard a particular piece of data belongs to. It can be a single column or a combination of columns.
2. Sharding Process:
    a. Choosing a Sharding Key:
        i. The sharding key is crucial as it determines how the data is distributed across shards. A good sharding key ensures even data distribution and efficient query performance.
        ii. Example: In a user database, a common sharding key could be the user ID.
    b. Data Distribution:
        i. The data is distributed across multiple shards based on the sharding key. This can be done using various strategies like range-based, hash-based, or list-based sharding.
3. Sharding Strategies:
    a. Range-Based Sharding:
        i. Divides data based on a range of values of the sharding key.
        ii. Example: Users with IDs 1-1000 in one shard, 1001-2000 in another.
        iii. Pros: Simple and intuitive.
        iv. Cons: Can lead to uneven distribution if data is not uniformly distributed.

```
-- Example: Range-based sharding for users
-- Shard 1: user_id 1-1000
CREATE TABLE shard1.users (user_id INT, name VARCHAR(255), ...);

-- Shard 2: user_id 1001-2000
CREATE TABLE shard2.users (user_id INT, name VARCHAR(255), ...);
```

    b. Hash-Based Sharding:
        i. Uses a hash function on the sharding key to determine the shard.
        ii. Example: shard = hash(user_id) % number_of_shards
        iii. Pros: Ensures even data distribution.
        iv. Cons: More complex to implement, potential challenges with shard rebalancing.

```
// Example: Hash-based sharding logic in Java
int numberOfShards = 4;
int shard = userId.hashCode() % numberOfShards;
```

   c. List-Based Sharding:
      i. Data is distributed based on a predefined list of values.
      ii. Example: User data from different regions can be placed in different shards.

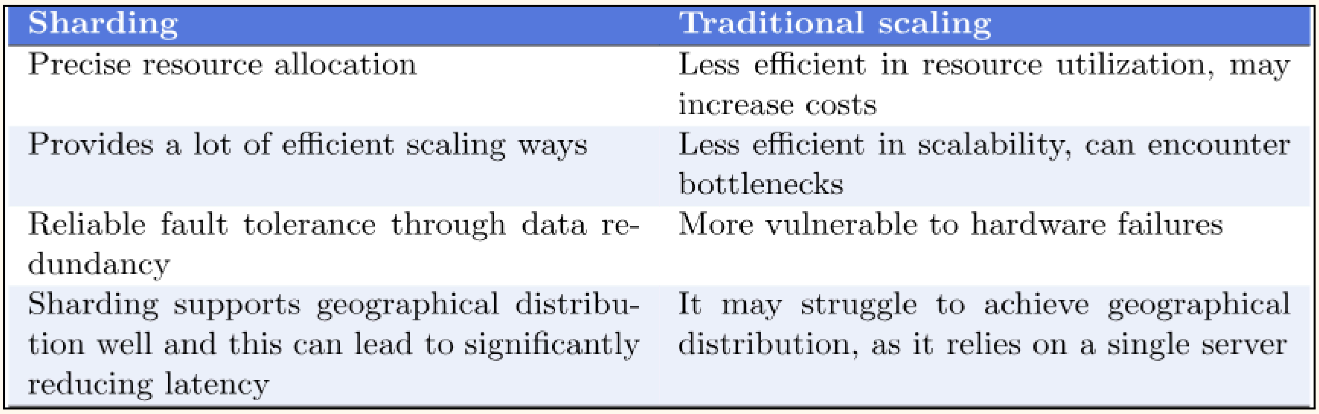
```
-- Example: List-based sharding for regional data
-- Shard 1: users from 'North America'
CREATE TABLE shard1.users (user_id INT, name VARCHAR(255), region VARCHAR(255), ...);

-- Shard 2: users from 'Europe'
CREATE TABLE shard2.users (user_id INT, name VARCHAR(255), region VARCHAR(255), ...);
```

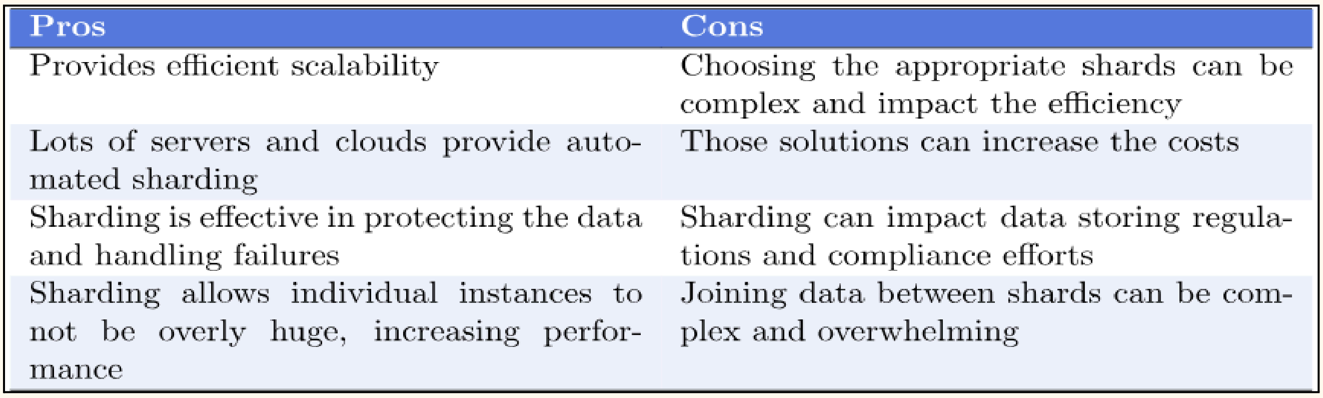
4. Query Routing:
   a. Query Router: A middleware layer that intercepts database queries and routes them to the appropriate shard based on the sharding key.
   b. Example: If a query involves user_id = 1234, the router calculates the shard using the sharding logic and directs the query to the correct shard.

```
// Example: Query routing logic in Java
public Shard getShard(int userId) {
   int shardId = userId.hashCode() % numberOfShards;
   return shardMap.get(shardId);
}
```

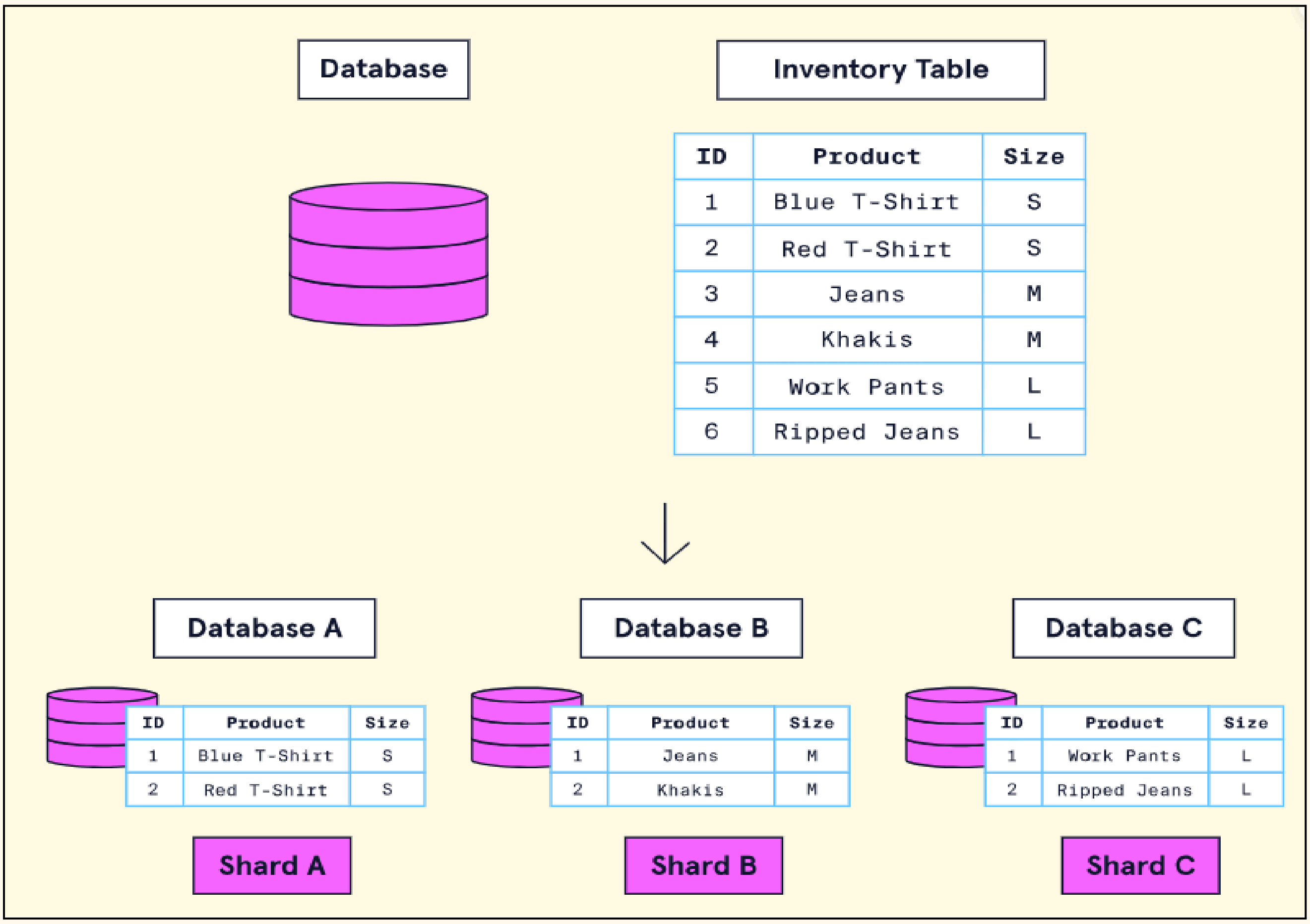
5. Balancing the Load:
   a. Sharding should distribute the data and query load evenly across all servers to prevent hotspots and ensure efficient resource utilization.



Sharding	Traditional scaling
Precise resource allocation	Less efficient in resource utilization, may increase costs
Provides a lot of efficient scaling ways	Less efficient in scalability, can encounter bottlenecks
Reliable fault tolerance through data redundancy	More vulnerable to hardware failures
Sharding supports geographical distribution well and this can lead to significantly reducing latency	It may struggle to achieve geographical distribution, as it relies on a single server



Pros	Cons
Provides efficient scalability	Choosing the appropriate shards can be complex and impact the efficiency
Lots of servers and clouds provide automated sharding	Those solutions can increase the costs
Sharding is effective in protecting the data and handling failures	Sharding can impact data storing regulations and compliance efforts
Sharding allows individual instances to not be overly huge, increasing performance	Joining data between shards can be complex and overwhelming



ID	Product	Size
1	Blue T-Shirt	S
2	Red T-Shirt	S
3	Jeans	M
4	Khakis	M
5	Work Pants	L
6	Ripped Jeans	L

ID	Product	Size
1	Blue T-Shirt	S
2	Red T-Shirt	S

ID	Product	Size
1	Jeans	M
2	Khakis	M

ID	Product	Size
1	Work Pants	L
2	Ripped Jeans	L

FAILOVER:

Failover is a critical concept in system design aimed at enhancing the reliability and availability of systems. It refers to the process by which a system automatically switches to a standby or backup component when the primary component fails. This ensures that services remain available and uninterrupted even in the case of hardware or software failures.

A set of computer servers that together provide continuous availability (CA), fault tolerance (FT), or high availability (HA) is called a failover cluster.

Key Components of Failover:

- Primary System: The main system or component that handles the normal operation.
- Secondary System (Backup/Standby): The system or component that takes over operations if the primary system fails.
- Heartbeat Mechanism: Regular checks to monitor the health of the primary system and detect failures.
- Failover Mechanism: The process or protocols that enable the switch to the secondary system.

How does Failover work:

- Heartbeat Monitoring:
    - Definition: Heartbeats are regular signals sent between the primary and secondary systems to confirm that the primary system is operational.
    - Implementation: Heartbeat signals are sent at regular intervals. If the secondary system does not receive a heartbeat within a specified timeout period, it assumes that the primary system has failed.
- Failure Detection:
    - Process: When the heartbeat mechanism detects that the primary system has failed, it triggers the failover process.
    - Mechanisms: Failure can be detected through missed heartbeats, error messages, or performance metrics falling below a threshold.

- Failover Activation:
    - Switching Roles: The secondary system activates, taking over the role of the primary system.
    - Data Synchronization: The secondary system should be in sync with the primary to ensure a seamless transition. Techniques like replication and regular backups help in keeping the data synchronized.
- Recovery and Failback:
    - Primary System Recovery: The primary system is repaired and brought back online.
    - Failback: The roles are switched back, with the primary system resuming its original role, and the secondary system returning to standby.

Types of Failover:

1. Cold Failover:
    a. Characteristics: The secondary system is activated only after the primary system fails and the failure is detected.
    b. Downtime: Some downtime is expected during the switch.
2. Warm Failover:
    a. Characteristics: The secondary system is running and periodically updated with the state of the primary system. It can quickly take over with minimal downtime.
    b. Synchronization: Regular synchronization with the primary system is required.
3. Hot Failover:
    a. Characteristics: The secondary system is fully operational and continuously synchronized with the primary system.
    b. Downtime: No downtime, as the switch is instantaneous.

Benefits of Failover:

1. High Availability: Ensures continuous availability of services even in the event of a failure.
2. Reliability: Enhances system reliability by providing a backup that can quickly take over.
3. Minimized Downtime: Reduces the downtime experienced during system failures.
4. Business Continuity: Supports business operations by maintaining service continuity and reducing the impact of failures.

Challenges and Considerations:

1. Data Synchronization: Ensuring that the secondary system has the latest data and state information.
2. Testing: Regularly testing the failover mechanism to ensure it works as expected.
3. Cost: Implementing and maintaining failover mechanisms can be costly.
4. Complexity: Adding failover mechanisms increases the complexity of system architecture and management.

PROXY SERVER:

A proxy server is a system that acts as an intermediary between you and the internet, providing a gateway that helps prevent cyber attackers from entering your private network. It serves as a buffer between your computer and the web pages you visit, ensuring a secure and efficient browsing experience.

When you use a proxy server, your internet requests are processed through it, keeping your IP address hidden and adding an extra layer of security to your connection. This is especially useful when connecting to public Wi-Fi networks or accessing sensitive data.

Types of Proxy Servers:

1. Forward Proxy:
    a. Sits in front of the client and forwards requests from the client to the internet.

    b. It is widely used to retrieve data from a range of sources.
    c. It acts on behalf of individual users and can help bypass web restrictions or enhance online privacy.
    d. Commonly used for content filtering, privacy, and access control.

2. Reverse Proxy:
    a. Sits in front of the server and forwards requests from clients to the server.
    b. It is typically used by websites to handle large numbers of simultaneous visitors, distributing the load among multiple servers providing additional security features.
    c. Commonly used for load balancing, caching, and improving security.
3. Transparent Proxy:
    a. Intercepts requests from clients without any configuration required on the client-side.
    b. Often used by ISPs and network administrators for monitoring and filtering traffic.
4. Anonymous Proxy:
    a. Hides the client's IP address and other identifying information from the destination server.
    b. Used to enhance privacy and anonymity online.
5. High-Anonymity Proxy (Elite Proxy):
    a. Provides the highest level of anonymity by masking the client's IP address and not revealing itself as a proxy server.

How does a Proxy Server work:

1. Client Request:
    a. The client sends a request to access a resource (e.g., a web page).
    b. Instead of going directly to the destination server, the request is routed to the proxy server.
2. Proxy Server Processing:
    a. The proxy server receives the client's request.
    b. Depending on its configuration, the proxy may modify the request, cache the response, or apply access control rules.
3. Request Forwarding:
    a. The proxy server forwards the client's request to the destination server.
    b. If caching is enabled and the requested resource is in the cache, the proxy server may return the cached response directly to the client, avoiding a round-trip to the destination server.
4. Server Response:
    a. The destination server processes the request and sends a response back to the proxy server.
5. Response Forwarding:
    a. The proxy server forwards the response from the destination server to the client.
    b. If caching is enabled, the proxy server may store the response for future requests.

Benefits of Proxy Servers:

1. Enhanced Security: By filtering traffic and hiding IP addresses, proxy servers can protect against attacks.
2. Improved Performance: Caching frequently accessed resources reduces load times and server strain.
3. Controlled Access: Proxies can enforce policies and restrict access to specific resources or websites.
4. Anonymity: By masking IP addresses, proxies enhance user privacy and anonymity online.

Challenges of Proxy Servers:

1. Configuration Complexity: Setting up and maintaining a proxy server can be complex and require ongoing management.
2. Latency: Additional hops in the network path can introduce latency.
3. Single Point of Failure: If not properly managed, a proxy server can become a single point of failure in the network.

WEBSOCKET:

WebSockets provide a full-duplex communication channel over a single, long-lived connection, allowing real-time data transfer between a client and a server. Unlike the traditional HTTP request/response model, WebSockets enable a persistent connection that remains open, facilitating continuous, bidirectional communication without the overhead of repeatedly opening and closing connections.

How does WebSocket work?

- Handshake:
  - The communication starts with an HTTP request from the client to the server, known as the WebSocket handshake.
  - The client requests to upgrade the connection from HTTP to WebSocket.
  - If the server supports WebSockets, it agrees to the upgrade, and the connection switches protocols.
- Data Exchange:
  - Once the handshake is complete, the client and server can send messages to each other independently.
  - This full-duplex communication allows data to be sent and received at any time by either party.
- Connection Termination:
  - The WebSocket connection remains open until either the client or the server decides to close it.
  - The connection can be closed gracefully by sending a close frame or abruptly by terminating the underlying TCP connection.

What are WebSockets Used For?

- WebSockets are used for real-time, event-driven communication between clients and servers. They are particularly useful for building software applications requiring instant updates, such as real-time chat, messaging, and multiplayer games.
- WebSockets establish a persistent connection between the client and the server. This means that once the connection is established, the client and the server can send data to each other at any time without continuous polling requests. This allows real time communication, where updates can be sent and received instantly.
- For example, when a user sends a message in a chat application, it can be instantly delivered to all other users without refreshing the page or making frequent HTTP requests. This results in a more seamless and efficient user experience.

Benefits:

- Low Latency: Provides near real-time communication with minimal latency compared to HTTP polling or long-polling.
- Efficiency: Reduces the overhead of establishing and closing connections repeatedly.
- Bidirectional Communication: Allows both the client and server to send messages independently and asynchronously.
- Persistent Connection: Maintains a continuous connection, which is ideal for applications requiring frequent data updates.

Use Cases:

- Real-Time Applications: Chat applications, live sports updates, stock price monitoring, multiplayer online games.
- Collaborative Tools: Real-time document editing, whiteboards, and collaborative coding platforms.
- IoT (Internet of Things): Communication between IoT devices and servers.
- Live Feeds: News feeds, social media updates, live event streaming.

Challenges:

- Scalability: Managing a large number of open connections can be challenging and resource-intensive.
- Firewall and Proxy Issues: Some firewalls and proxies may block WebSocket connections.
- Complexity: Implementing and debugging WebSocket applications can be more complex compared to traditional HTTP-based communication.
- Security: Ensuring secure communication over WebSockets requires attention to potential vulnerabilities such as cross-site WebSocket hijacking.

WebSocket Connection vs HTTP Connection:

WebSocket Connection	HTTP Connection
WebSocket is a bidirectional communication protocol that can send the data from the client to the server or from the server to the client by reusing the established connection channel. The connection is kept alive until terminated by either the client or the server.	The HTTP protocol is a unidirectional protocol that works on top of TCP protocol which is a connection-oriented transport layer protocol, we can create the connection by using HTTP request methods after getting the response HTTP connection get closed.
Almost all the real-time applications like (trading, monitoring, notification) services use WebSocket to receive the data on a single communication channel.	Simple RESTful application uses HTTP protocol which is stateless.
All the frequently updated applications used WebSocket because it is faster than HTTP Connection.	When we do not want to retain a connection for a particular amount of time or reuse the connection for transmitting data; An HTTP connection is slower than WebSockets.

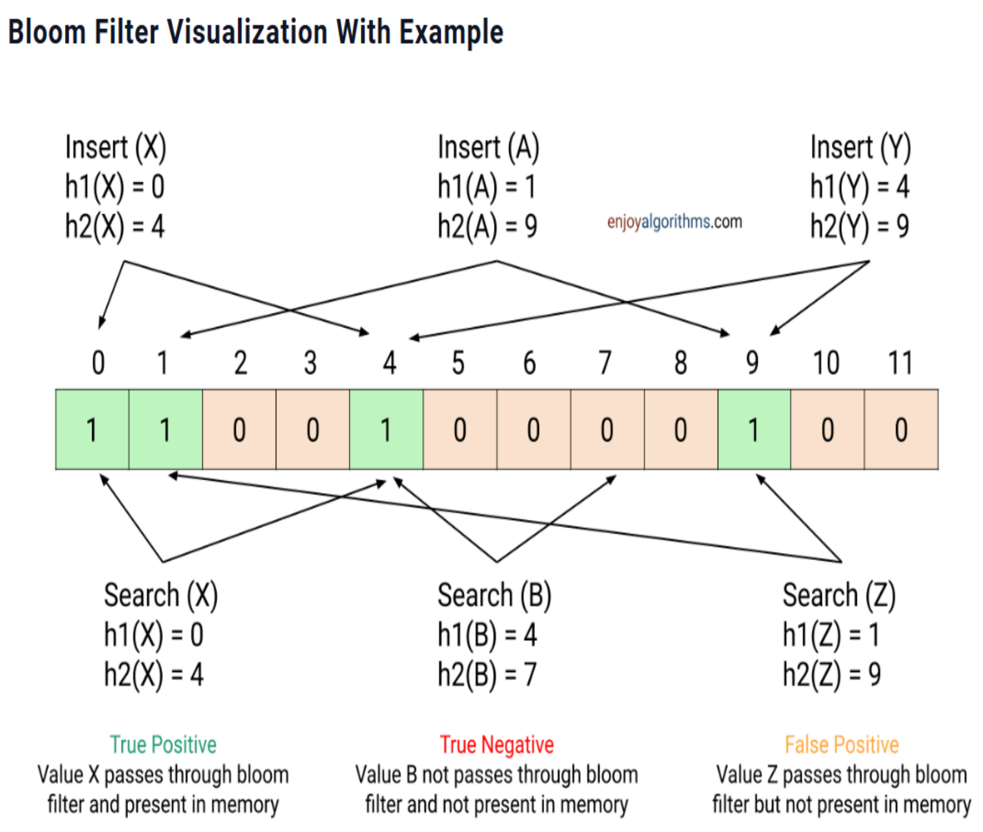
BLOOM FILTERS:

A Bloom filter is a probabilistic data structure that is used to test whether an element is a member of a set. It is highly space-efficient and allows for fast insertion and query operations. However, it has a certain probability of false positives, meaning that it can incorrectly indicate that an element is in the set when it is not, but it never has false negatives.

How does Bloom Filters work?

- Initialization:
    - A Bloom filter is initialized with a bit array of m bits, all set to 0.
    - It also uses k different hash functions, each of which maps an input to one of the m array positions uniformly.
- Adding an Element:
    - To add an element x to the Bloom filter, it is hashed using the k hash functions to get k positions.
    - The bits at all k positions in the array are set to 1.

- Querying an Element:
    - To check if an element y is in the set, it is hashed using the same k hash functions to get k positions.
    - If any of the bits at these k positions is 0, the element is definitely not in the set.
    - If all bits at these k positions are 1, the element is possibly in the set (with some probability of being a false positive).



Example:
Initialization:
Bit array: [0, 0, 0, 0, 0, 0, 0, 0] (size m = 8)
Hash functions: h1(x), h2(x) (where k = 2)
Adding elements "cat" and "dog":
"cat":
h1("cat") = 3
h2("cat") = 5
Bit array: [0, 0, 0, 1, 0, 1, 0, 0]
"dog":
h1("dog") = 2
h2("dog") = 5
Bit array: [0, 0, 1, 1, 0, 1, 0, 0]
Querying elements "cat", "dog", "fish":
"cat":
h1("cat") = 3
h2("cat") = 5
Bits at positions 3 and 5 are both 1: possibly in the set.
"dog":
h1("dog") = 2
h2("dog") = 5
Bits at positions 2 and 5 are both 1: possibly in the set.
"fish":
h1("fish") = 1
h2("fish") = 4
Bit at position 4 is 0: definitely not in the set.

Benefits:

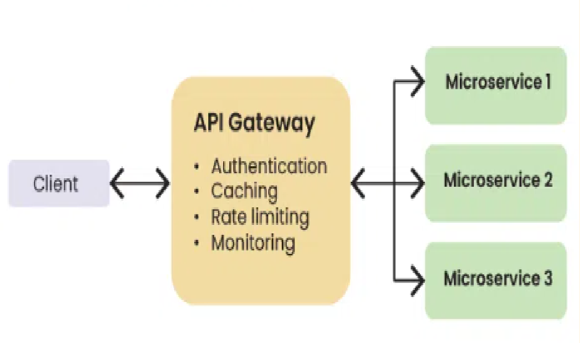
- Space Efficiency: Uses significantly less memory compared to storing the actual elements or even their hashes.
- Speed: Both insertion and query operations are very fast (constant time complexity, O(k)).

Drawbacks:

- False Positives: Cannot definitively say an element is in the set, only that it might be. This can be mitigated by choosing an appropriate size for the bit array and the number of hash functions.
- Not Removable: Standard Bloom filters do not support removal of elements since it could lead to false negatives.
- Applications

- Database Query Optimization: Quickly check if a key might exist in a database to avoid unnecessary disk reads.
- Network Security: Used in distributed systems for cache filtering, preventing the need to query all nodes for non-existent data.
- Web Cache: Used to filter requests to a web cache to see if the requested data is possibly stored there.

API GATEWAY:



An API Gateway is a server that acts as an API front-end, receiving API requests, enforcing throttling and security policies, passing requests to the back-end service, and then passing the response back to the requester. It is a critical component in microservices architectures and modern cloud-native applications.

API Gateway Capabilities:

- Security policy – Authentication, authorization, access control, and encryption
- Routing policy – Routing, rate limiting, request/response manipulation, circuit breaker, blue-green and canary deployments, A/B testing, load balancing, health checks, and custom error handling
- Observability policy – Real-time and historical metrics, logging, and tracing.

Key Features of an API Gateway:

- **Request Routing:** Routes client requests to the appropriate microservice based on the request path or other request data.
- **Load Balancing:** Distributes incoming requests across multiple instances of a service to ensure no single instance becomes overwhelmed.
- **Authentication and Authorization:** Handles user authentication and enforces authorization policies to control access to different services.
- **Rate Limiting and Throttling:** Controls the rate of incoming requests to prevent abuse and ensure fair usage among clients.
- **Logging and Monitoring:** Captures detailed logs of API calls and performance metrics for monitoring and troubleshooting purposes.
- **Caching:** Stores responses from back-end services temporarily to reduce load and improve response times for repeated requests.
- **Transformation and Aggregation:** Modifies request and response payloads (e.g., format conversion) and aggregates responses from multiple services.

How an API Gateway Works:

- **Client Request:** The client sends a request to the API Gateway instead of directly communicating with the individual microservices.
- **Processing by API Gateway:** The API Gateway processes the request, which may include tasks like authentication, authorization, rate limiting, and logging.
- **Routing to Appropriate Service:** The API Gateway routes the request to the appropriate microservice based on the request details.
- **Service Response:** The microservice processes the request and sends the response back to the API Gateway.
- **Response Handling:** The API Gateway processes the response, possibly modifying it (e.g., adding headers, transforming the payload) before sending it back to the client.

Example Use Case:

Consider a simplified e-commerce application with the following microservices:

- User Service: Manages user accounts.
- Product Service: Manages product information.
- Order Service: Handles orders and payments.

Without API Gateway:

- The client needs to know the endpoints for each microservice.
- The client makes direct requests to each microservice.
- Each microservice independently handles cross-cutting concerns like authentication, logging, etc.

With API Gateway:

- The client makes requests to a single endpoint (the API Gateway).
- The API Gateway routes the requests to the appropriate microservice.
- Cross-cutting concerns are centralized in the API Gateway.

Benefits of an API Gateway:

- Simplifies Client Code: Clients interact with a single endpoint rather than multiple microservice endpoints.
- Centralized Cross-Cutting Concerns: Centralizes functionality like authentication, logging, and rate limiting, reducing redundancy.
- Improved Security: Enforces security policies at a single point, reducing the attack surface.
- Load Balancing and Failover: Distributes traffic and provides failover capabilities to improve availability and reliability.

Challenges and Considerations:

- Single Point of Failure: The API Gateway itself can become a bottleneck or point of failure if not properly managed.
- Latency: Adds an extra hop in the request-response cycle, which can introduce latency.
- Complexity: Managing and configuring an API Gateway can add complexity to the system architecture.

How differently API Gateway works with Microservices and Monolithic Architecture?

Aspect	Monolithic Architecture	Microservices Architecture
Request routing	In a monolithic architecture, the API Gateway typically routes requests to different parts of the monolith based on the request URL or other criteria	In a microservices architecture, the API Gateway routes requests to different microservices based on the request URL or other criteria, acting as a kind of “front door” to the microservices ecosystem.
Service discovery	In a monolithic architecture, service discovery is not typically a concern, as all parts of the application are contained within the same codebase.	In a microservices architecture, the API Gateway may need to use service discovery mechanisms to dynamically locate and route requests to the appropriate microservices.

Authentication and authorization	In both architectures, the API Gateway can handle authentication and authorization.	However, in a microservices architecture, there may be more complex authorization scenarios, as requests may need to be authorized by multiple microservices.
Load balancing	In both architectures, the API Gateway can perform load balancing.	However, in a microservices architecture, load balancing may be more complex, as requests may need to be load balanced across multiple instances of multiple microservices.
Fault tolerance	In both architectures, the API Gateway can provide fault tolerance by retrying failed requests and routing requests to healthy instances of services.	However, fault tolerance may be more critical in a microservices architecture, where the failure of a single microservice should not bring down the entire system.

CHECKSUM:

A checksum is a small-sized piece of data derived from another block of digital data for the purpose of detecting errors that may have been introduced during its transmission or storage. Checksums are typically used to verify the integrity of data.

How Does a Checksum Work?

1. Data Input: The original data, which could be a file, a message, or any data block, is fed into a checksum algorithm.
2. Checksum Algorithm: The algorithm processes the input data and produces a fixed-size string or integer that uniquely represents the original data. This value is the checksum.
3. Data Transmission: The data along with its checksum is sent to the receiver. The receiver computes the checksum on the received data using the same algorithm.
4. Verification: The receiver compares the computed checksum with the transmitted checksum. If they match, the data is considered intact. If not, an error is detected, indicating potential data corruption or tampering.

Types of Checksums:

- Parity Bit: A parity bit is a single bit that is added to a group of bits to make the total number of 1s either even (even parity) or odd (odd parity). While it can detect single bit errors, it fails if an even number of bits are flipped.
- Modular Sum: A basic form where data bytes are added together using modulo arithmetic.
- CRC (Cyclic Redundancy Check): It works by treating the data as a large binary number and dividing it by a predetermined divisor. The remainder of this division becomes the checksum. CRCs are designed to detect common errors caused by noise in transmission channels.
- Cryptographic Hash Functions: These are one-way functions that generate a fixed-size hash value from the data. Popular examples include MD5, SHA-1, and SHA-256.

Role of checksum algorithms in data integrity:

- Error Detection: Checksum algorithms generate a unique hash value for a given set of data. By comparing this hash value before and after transmission or storage, it’s possible to detect errors such as data corruption, tampering, or transmission errors. If the hash values do not match, it indicates that the data has been altered in some way.

- Data Verification: Checksums can be used to verify that data has been received or stored correctly. By recalculating the checksum on the received data and comparing it to the transmitted checksum, the recipient can ensure that the data has not been altered during transmission.
- Data Integrity Assurance: Checksums provide a way to verify the integrity of data without needing to compare the entire dataset. This is particularly useful for large files or data streams, where verifying each individual bit would be impractical.
- Efficiency: Checksum algorithms are computationally efficient, allowing for quick verification of data integrity without significant overhead.

Applications of Checksums:

- Data Transmission: Used to verify the integrity of data transferred over networks (e.g., TCP/IP packets).
- Data Storage: Ensures the integrity of stored data, particularly in RAID systems and file systems.
- Software Distribution: Verifies the integrity of downloaded software packages to prevent corruption or tampering.
- Digital Signatures: Helps in verifying the authenticity and integrity of digital documents.

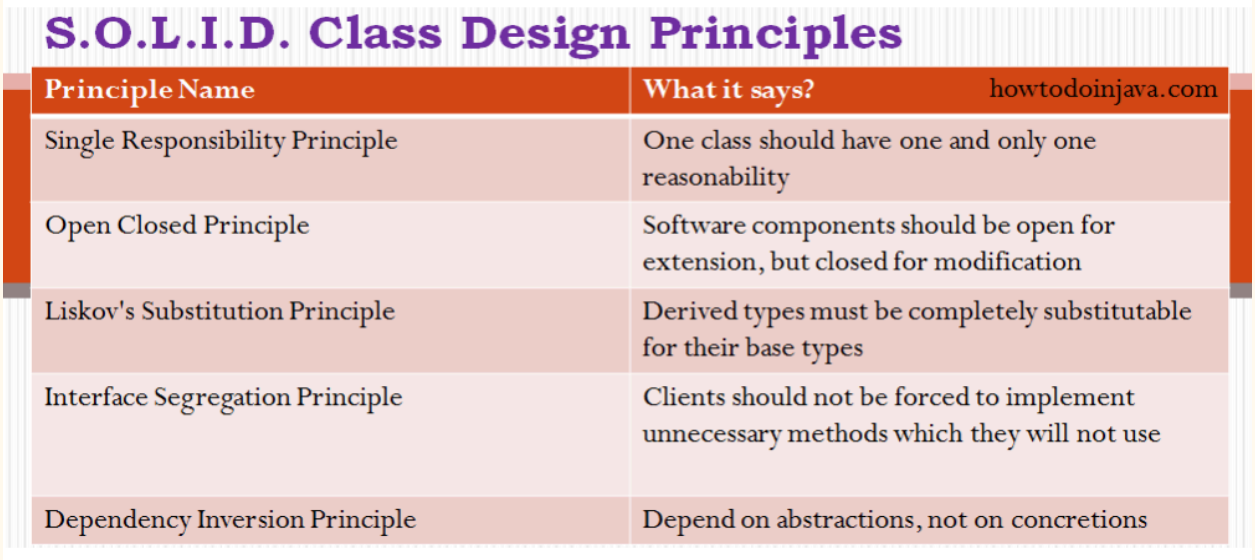
Benefits of Using Checksums:

- Error Detection: Efficiently detects errors in data transmission and storage.
- Data Integrity: Ensures that data has not been altered or corrupted.
- Lightweight: Checksums are typically small and easy to compute, making them suitable for various applications.

Limitations:

- False Positives: While rare, different data can produce the same checksum (collision), leading to false positives.
- Limited Error Correction: Checksums primarily detect errors but do not correct them.
- Security: Basic checksums (like simple parity or modular sums) are not suitable for cryptographic security needs as they are easy to forge.

SOLID PRINCIPAL:
https://medium.com/@javatechie/solid-design-principle-java-ae96a48db97



S.O.L.I.D. Class Design Principles

Principle Name	What it says? howtodoinjava.com
Single Responsibility Principle	One class should have one and only one reasonability
Open Closed Principle	Software components should be open for extension, but closed for modification
Liskov's Substitution Principle	Derived types must be completely substitutable for their base types
Interface Segregation Principle	Clients should not be forced to implement unnecessary methods which they will not use
Dependency Inversion Principle	Depend on abstractions, not on concretions

1. **Single Responsibility Principle:** This principle states that “a class should have only one reason to change” which means every class should have a single responsibility or single job or single purpose.

Most of the time it happens that when programmers have to add features or new behavior they implement everything into the existing class which is completely wrong. It makes their code lengthy, complex and consumes time when later something needs to be modified. Use layers in your application and break God classes into smaller classes or modules.

The key idea behind SRP is to promote high cohesion within classes, meaning that the members (methods and fields) of a class should be closely related and work together to achieve a single, well-defined purpose. This principle encourages a modular and maintainable design by avoiding classes that are burdened with multiple, unrelated responsibilities.

Some key points related to the Single Responsibility Principle:

1) Separation of Concerns: SRP promotes the separation of concerns by ensuring that each class or module is responsible for a single aspect of the system's functionality. This makes it easier to understand, maintain, and modify code.
2) Reason to Change: A "reason to change" refers to a potential modification in the future. If a class has multiple responsibilities, a change in one area might affect other areas, leading to increased complexity and the likelihood of introducing bugs.
3) High Cohesion: High cohesion means that the members of a class are closely related and work together to achieve a common goal. Classes with high cohesion are more focused and easier to reason about.

2. **Open/Closed Principle:** This principle states that “software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification” which means you should be able to extend a class behavior, without modifying it.

Suppose developer A needs to release an update for a library or framework and developer B wants some modification or add some feature on that then developer B is allowed to extend the existing class created by developer A but developer B is not supposed to modify the class directly. Using this principle separates the existing code from the modified code so it provides better stability, maintainability and minimizes changes as in your code.

Key points related to the Open/Closed Principle:

1) Open for Extension: This part of the principle encourages the design of software components that can be extended to accommodate new behaviors or features. It should be possible to add new functionality without altering the existing code.
2) Closed for Modification: Once a module is stable and has been released, its source code should be closed for modification. Existing functionality should not be changed, minimizing the risk of introducing bugs in already working code.
3) Use of Abstraction: The Open/Closed Principle often involves the use of abstraction, such as interfaces or abstract classes. The idea is to define a stable interface that can be extended by creating new implementations.

3. **Liskov’s Substitution Principle:** The principle was introduced by Barbara Liskov in 1987 and according to this principle “Derived or child classes must be substitutable for their base or parent classes“. This principle ensures that any class that is the child of a parent class should be usable in place of its parent without any unexpected behavior.

In simpler terms, Subtypes must be substitutable for their base types without altering the correctness of the program. If a class (subtype) is a derived class of another class (base type), instances of the derived class should be able to replace instances of the base class without affecting the correctness of the program.

few guidelines to adhere to the Liskov Substitution Principle -

1) Behavioral Compatibility: Subtypes must exhibit behavioral compatibility with the base type. This means that the derived class should provide the same behavior as the base class, or a behavior that is compatible with the base class.

2) No Weakening of Preconditions: The preconditions (requirements) of the base class should not be weakened in the derived class. The derived class can strengthen preconditions, but it should not relax them.
3) No Strengthening of Postconditions: The postconditions (guarantees) of the base class should not be strengthened in the derived class. The derived class can weaken postconditions, but it should not make stronger guarantees.

4. **Interface Segregation Principle:** This principle is the first principle that applies to Interfaces instead of classes in SOLID and it is similar to the single responsibility principle. It states that “do not force any client to implement an interface which is irrelevant to them“. Here your main goal is to focus on avoiding fat interfaces and give preference to many small client-specific interfaces.

Key points related to the Interface Segregation Principle:

1) Specific Interfaces: Instead of having a large, general-purpose interface, it is better to create specific interfaces that are tailored to the needs of the clients that use them.
2) Client-Specific Interfaces: Clients should not be forced to implement methods they do not need. If a client is interested in a subset of the functionality provided by an interface, it should not be obligated to provide implementations for the entire interface.
3) Avoiding Fat Interfaces: Having large interfaces with many methods can lead to "fat interfaces," where clients are burdened with implementing methods they might not need. This violates the principle of least astonishment and makes the code less maintainable.

5. **Dependency Inversion Principle:** The Dependency Inversion Principle (DIP) is one of the SOLID principles of object-oriented design. It was introduced by Robert C. Martin to address the issue of high-level modules being dependent on low-level modules, and both being dependent on details.

The formal definition of the Dependency Inversion Principle is as follows: High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

In simpler terms, the DIP encourages the use of abstractions (interfaces or abstract classes) to define the contracts between high-level and low-level modules. High-level modules and low-level modules should both depend on these abstractions, rather than on concrete implementations

Here are some key points related to the Dependency Inversion Principle:

1) Abstractions : Abstractions are used to define the contracts or interfaces that high-level and low-level modules should adhere to. They represent a higher-level view of what the modules should provide.
2) Dependency Inversion: Instead of high-level modules depending directly on low-level modules, both should depend on abstractions. This inversion of dependencies allows for more flexibility in the system.
3) Decoupling: By depending on abstractions, the coupling between high-level and low-level modules is reduced. Changes in low-level modules do not directly impact high-level modules, promoting a more modular and maintainable design.
4) Details Depending on Abstractions: Concrete implementations (details) should depend on the abstractions. This means that classes providing specific implementations should adhere to the contracts defined by the abstractions.

Behavioral	Creational	Structural

Iterator Pattern	Factory Pattern	Adapter Pattern
Interpreter Pattern	Abstract Factory Pattern	Bridge Pattern
Mediator Pattern	Singleton Pattern	Composite Pattern
Memento Pattern	Prototype Pattern	Decorator Pattern
Observer Pattern	Builder Pattern	Facade Pattern
State Pattern	Object Pool	Flyweight Pattern
Strategy Pattern		Proxy Pattern
Template Pattern		
Visitor Pattern		

SINGLETON DESIGN PATTERN:

Singleton Design Pattern | Implementation - GeeksforGeeks
Singleton Design Pattern_sudoCODE | Singleton Design Pattern_Geekific
Singleton Design Pattern is a creational design pattern that ensures a class has only one instance and provides a global point of access to it. This pattern is particularly useful when exactly one object is needed to coordinate actions across the system.

Key Concept of Singleton Method:

- Private Constructor: The Singleton class has a private constructor to prevent the instantiation of the class from external entities.
- Private Instance: The class contains a private static instance of itself.
- Static Method: A static method is provided to access the instance, and it ensures that only one instance is created if it doesn't exist.

Use Case of Singleton:

- **Database Connection Management:** Ensuring a single connection instance throughout the application.
- **Logger Classes:** Managing a single logger instance to centralize log information.

Advantages:

1. **Single Instance:**
    a. Advantage: Ensures that a class has only one instance.
    b. Explanation: This avoids unnecessary instantiation and guarantees that there's a single point of access to that instance.
2. **Global Point of Access:**
    a. Advantage: Provides a global point of access to the instance.
    b. Explanation: Any part of the application can access the Singleton instance, promoting a centralized and easily reachable point for shared resources.
3. **Lazy Initialization:**

    a. Advantage: Supports lazy initialization.
    b. Explanation: The instance is created only when it is needed for the first time, improving resource efficiency.
4. **Efficient Resource Management:**
    a. Advantage: Efficiently manages resources.
    b. Explanation: In scenarios where creating multiple instances is resource-intensive, the Singleton pattern ensures optimal usage of resources.
5. **Prevents Duplicate Configuration:**
    a. Advantage: Helps prevent duplicate configuration settings.
    b. Explanation: In scenarios where a single configuration should be maintained throughout the application, Singleton ensures that only one configuration instance exists.

Disadvantages:

1. **Global State:**
    a. Disadvantage: Can introduce a global state in the application.
    b. Explanation: As a single instance is accessible globally, changes to its state can affect the entire application, potentially leading to unintended consequences.
2. **Testing Challenges:**
    a. Disadvantage: Singleton can make unit testing challenging.
    b. Explanation: Since the Singleton instance is globally accessible, it can be difficult to isolate and test individual components independently.
3. **Violates Single Responsibility Principle:**
    a. Disadvantage: Violates the Single Responsibility Principle.
    b. Explanation: The Singleton class is responsible for both managing its instance and its primary functionality, which can lead to a lack of separation of concerns.
4. **Limited Extensibility:**
    a. Disadvantage: Can hinder extensibility.
    b. Explanation: Introducing subclasses or alternative implementations of the Singleton can be challenging without modifying the existing code.
5. **Potential for Unintended Consequences:**
    a. Disadvantage: Changes to the Singleton instance can impact the entire application.
    b. Explanation: Modifying the state of the Singleton may have unintended consequences in other parts of the system due to its global accessibility.

FACTORY DESIGN PATTERN:

Factory Method Design Pattern defines an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclass. The Factory Design Pattern is a creational design pattern that provides an interface for creating instances of a class, but allows subclasses to alter the type of objects that will be created. It promotes loose coupling by abstracting the process of object creation.

Components of the Factory Design Pattern:

Product: The interface or abstract class that defines the type of objects the factory method creates.
ConcreteProduct: The class that implements the Product interface.
Creator: The abstract class or interface that declares the factory method. It may also include other methods that work with products created by the factory method.
ConcreteCreator: The class that implements the Creator interface and overrides the factory method to produce instances of ConcreteProduct.

When to use Factory Method Design Pattern in Java?

Factory method design pattern can be used in java in following cases:

- A class cannot predict the type of objects it needs to create.
- A class wants its subclasses to specify the objects it creates.
- Classes delegate responsibility to one of multiple helper subclasses, and you aim to keep the information about which helper subclass is the delegate within a specific scope or location.

Use Cases of the Factory Method Design Pattern in Java:

- **Creational Frameworks:** JDBC (Java Database Connectivity) uses factories extensively for creating connections, statements, and result sets. Dependency injection frameworks like Spring and Guice rely heavily on factories to create and manage beans.
- **GUI Toolkits:** Swing and JavaFX use factories to create UI components like buttons, text fields, and labels, allowing for customization and flexibility in UI design.
- **Logging Frameworks:** Logging frameworks like Log4j and Logback use factories to create loggers with different configurations, enabling control over logging levels and output destinations.
- **Serialization and Deserialization:** Object serialization frameworks often use factories to create objects from serialized data, supporting different serialization formats and versioning.
- **Plugin Systems:** Plugin-based systems often use factories to load and create plugin instances dynamically, allowing for extensibility and customization.
- **Game Development:** Game engines often use factories to create different types of game objects, characters, and levels, promoting code organization and flexibility.
- **Web Development:**Web frameworks sometimes use factories to create view components, controllers, and services, enabling modularity and testability in web applications.

Advantages of Factory Method Design Pattern in Java:

- **Decoupling:** It separates object creation logic from the client code that uses those objects. This makes the code more flexible and maintainable because changes to the creation process don't require modifications to client code.
- **Extensibility:** It's easy to introduce new product types without changing the client code. You simply need to create a new Concrete Creator subclass and implement the factory method to produce the new product.
- **Testability:** It simplifies unit testing by allowing you to mock or stub out product creation during tests. You can test different product implementations in isolation without relying on actual object creation.
- **Code Reusability:** The factory method can be reused in different parts of the application where object creation is needed. This promotes centralizing and reusing object creation logic.
- Encapsulation: It hides the concrete product classes from the client code, making the code less dependent on specific implementations. This improves maintainability and reduces coupling.

Disadvantages of Factory Method Design Pattern in Java:

- **Increased Complexity:** It introduces additional classes and interfaces, adding a layer of abstraction that can make the code more complex to understand and maintain, especially for those unfamiliar with the pattern.
- **Overhead:** The use of polymorphism and dynamic binding can slightly impact performance, although this is often negligible in most applications.
- **Tight Coupling Within Product Hierarchies:** Concrete Creators are still tightly coupled to their corresponding Concrete Products. Changes to one often necessitate changes to the other.

- **Dependency on Concrete Subclasses:** The client code still depends on the abstract Creator class, requiring knowledge of its concrete subclasses to make correct factory method calls.
- **Potential for Overuse:** It's important to use the Factory Method pattern judiciously to avoid over-engineering the application. Simple object creation can often be handled directly without the need for a factory.
- **Testing Challenges:** Testing the factory logic itself can be more complex.

ABSTRACT FACTORY DESIGN PATTERN:

Abstract Factory Method is a creational design pattern, it provides an interface for creating families of related or dependent objects without specifying their concrete classes. Abstract factory pattern implementation provides us with a framework that allows us to create objects that follow a general pattern. So at runtime, the abstract factory is coupled with any desired concrete factory which can create objects of the desired type.

Components of the Abstract Factory Design Pattern:

- **AbstractFactory:** Declares an interface for operations that create abstract product objects.
- **ConcreteFactory:** Implements the operations declared in the AbstractFactory to create concrete product objects.
- **Product:** Defines a product object to be created by the corresponding concrete factory and implements the AbstractProduct interface.
- **Client:** Uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Use Cases:

1. **Cross-Platform GUI Libraries:** Abstract Factory can be used to create families of UI components (buttons, windows, text boxes) for different operating systems.
2. **Database Access Libraries:** For database access, abstract factory can be used to create families of database-related objects (connections, statements, result sets) for different database systems.
3. **Hardware Abstraction Libraries:** In systems interacting with hardware, abstract factory can be used to create families of objects representing specific hardware devices.

Advantages:

1. **Loose Coupling:** The client code is decoupled from the specific classes of objects it creates, allowing for flexibility in switching between different families of products.
2. **Consistent Families:** Abstract Factory ensures that the created objects are compatible and belong to the same family, as they are all created by the same factory.
3. **Scalability:** It is easy to introduce new families of products by adding new concrete factories and products without modifying existing client code.
4. **Isolation of concrete classes:** The Abstract Factory pattern helps you control the classes of objects that an application creates. Because a factory encapsulates the responsibility and the process of creating product objects, it isolates clients from implementation classes. Clients manipulate instances through their abstract interfaces. Product class names are isolated in the implementation of the concrete factory; they do not appear in client code.
5. **Exchanging Product Families easily:** The class of a concrete factory appears only once in an application, that is where it's instantiated. This makes it easy to change the concrete factory an application uses. It can use various product configurations simply by changing the concrete factory. Because an abstract factory creates a complete family of products, the whole product family changes at once.
6. **Promoting consistency among products:** When product objects in a family are designed to work together, it's important that an application uses objects from only one family at a time. AbstractFactory makes this easy to enforce.

Disadvantages:

1. **Complexity:** The introduction of new families of products can lead to an increase in the number of classes and interfaces, making the system more complex.
2. **Extensibility Challenges:** Extending the abstract factory pattern to support new types of products may require modifying existing interfaces, which can be challenging in large systems. That's because the AbstractFactory interface fixes the set of products that can be created. Supporting new kinds of products requires extending the factory interface, which involves changing the AbstractFactory class and all of its subclasses.

When to use

- When the system must be independent in terms of how its objects are created, composed, and represented.
- This constraint must be enforced when a family of related objects needs to be used together.
- When you want to provide an object library that does not show implementations but only exposes interfaces.
- When the system has to be configured with one of several families of objects.

Differences between Factory Method and Abstract Factory:

The Factory Method pattern defines an interface for creating objects, but it allows subclasses to decide which class to instantiate. It encapsulates the object creation logic in a separate method, which can be overridden by subclasses to create different types of objects. This pattern is useful when we want to create objects that belong to a single family of classes and defer instantiation to their subclasses.

On the other hand, the Abstract Factory pattern gives us a way to make groups of related or dependent objects without having to specify their concrete classes. It encapsulates a group of factories, each of which creates a family of objects that are related to each other.

- The main difference between a "factory method" and an "abstract factory" is that the factory method is a single method, and an abstract factory is an object.
- The factory method is just a method, it can be overridden in a subclass, whereas the abstract factory is an object that has multiple factory methods on it.
- The Factory Method pattern uses inheritance and relies on a subclass to handle the desired object instantiation.

STRATEGY DESIGN PATTERN:

Strategy pattern is a behavioral design pattern that allows the behavior of an object to be selected at runtime. The Strategy pattern is based on the idea of encapsulating a family of algorithms into separate classes that implement a common interface. The pattern consists of three main components: the Context, the Strategy, and the Concrete Strategy.

- The Context is the class that contains the object whose behavior needs to be changed dynamically.
- The Strategy is the interface or abstract class that defines the common methods for all the algorithms that can be used by the Context object.
- The Concrete Strategy is the class that implements the Strategy interface and provides the actual implementation of the algorithm.

Key Concept of Strategy Method Design Pattern:

Context: The context is the class that contains a reference to the strategy interface. It is the class that delegates the algorithm responsibilities to the strategy.

Strategy Interface: The strategy interface defines a set of methods that encapsulate the different algorithms. All concrete strategy classes must implement this interface to provide their specific implementations.

Concrete Strategies: These are the actual algorithm implementations. Each concrete strategy class implements the strategy interface, providing a specific behavior or algorithm.

Client: The client is responsible for selecting a concrete strategy and setting it in the context. It can switch between different strategies dynamically at runtime.

Context-Strategy Interaction: The context class contains a reference to a strategy object and delegates the algorithm responsibilities to the strategy. The context class does not implement the algorithm itself but relies on the strategy interface.

Dynamic Strategy Switching: One of the key features of the Strategy Method Pattern is the ability to switch between different algorithms dynamically at runtime. The client can change the strategy associated with the context without altering its structure.

Encapsulation of Algorithms: Each algorithm is encapsulated within its own strategy class, promoting code reusability and maintainability. Changes to one algorithm (strategy) do not affect the other algorithms, as they are independent and encapsulated.

Flexibility and Extensibility: The Strategy Method Pattern allows for a high degree of flexibility in selecting algorithms. It is easy to add new strategies without modifying existing code, making the system extensible.

Promotes Code Reusability: Since each algorithm is encapsulated in its own strategy class, they can be reused in different contexts or scenarios.

Advantages:

Flexibility: The Strategy pattern allows the behavior of an object to be changed dynamically at runtime by selecting different algorithms.

Modularity: The pattern encapsulates the algorithms in separate classes, making it easy to add or remove algorithms without affecting other parts of the code.

Testability: The pattern makes it easy to test the different algorithms separately, without affecting the overall behavior of the code.

Open-Closed Principle: The Strategy pattern follows the Open-Closed Principle, which states that a class should be open for extension but closed for modification.

Use cases of Strategy Method Design Pattern in Java:

1. **File Format Converters:** Suppose we have an application that needs to convert data between different file formats (e.g., CSV to JSON, XML to YAML). We can use the Strategy Pattern to define a family of algorithms for each conversion strategy.
2. **Algorithm Variations:** Sorting Algorithms: If we have a sorting algorithm that needs to be interchangeable (e.g., bubble sort, quicksort, mergesort), we can use the Strategy Pattern to encapsulate each sorting algorithm as a separate strategy.
3. **Searching Algorithms:** Similar to sorting, we can have different search algorithms (linear search, binary search, etc.) encapsulated as strategies.
4. **Payment Gateways:** In a payment processing system, you might have different payment gateways (e.g., PayPal, Stripe, Square). Using the Strategy Pattern, we can encapsulate the logic for each payment gateway as a separate strategy, allowing the system to switch between different payment methods easily.
5. **Compression Algorithms:** If we are working with a system that involves compressing and decompressing data, different compression algorithms (e.g., gzip, zlib, lzma) can be implemented as separate strategies.
6. **Image Processing:** In an image processing application, we may have different algorithms for image filtering, resizing, or color adjustments. Each of these algorithms can be implemented as a separate strategy, allowing the user to choose the desired image processing technique.

7. **Game Development:** In game development, strategies can be used for different behaviors of characters or enemies. For example, we might have different strategies for the movement, attack, and defense of game entities.
8. **Network Communication Protocols:** In networking, different communication protocols (e.g., TCP, UDP) can be implemented as separate strategies. This allows the system to switch between protocols without affecting the overall communication logic.
9. **Logging Strategies:** Logging mechanisms in software can have different strategies for output (e.g., console logging, file logging, database logging). The Strategy Pattern can be applied to switch between logging strategies dynamically.

Advantage of Strategy Method design Pattern in Java:

1. **Flexibility and Extensibility:** Strategies are encapsulated in separate classes, making it easy to add, remove, or modify strategies without affecting the client code. New algorithms can be introduced without altering the existing codebase, promoting an open/closed principle.
2. **Clean Code and Separation of Concerns:** The pattern promotes clean code by separating the concerns of the algorithm from the client code. Each strategy is encapsulated in its own class, leading to better code organization and maintenance.
3. **Promotes Code Reusability:** Strategies can be reused in different contexts, as they are independent and encapsulated in their own classes. Once a new strategy is implemented, it can be easily reused in other parts of the application without duplicating code.
4. **Easy Testing:** Testing is simplified because each strategy is a separate class. It is easier to write unit tests for individual strategies, ensuring that each algorithm behaves as expected.
5. **Dynamic Runtime Behavior:** The pattern allows clients to switch between different strategies at runtime. This flexibility is valuable when the algorithm to be used is determined dynamically during the program's execution.
6. **Encourages Single Responsibility Principle:** Each strategy class has a single responsibility: encapsulating a specific algorithm. This aligns with the Single Responsibility Principle, making the codebase more maintainable and understandable.
7. **Decouples Algorithms from Context:** The pattern decouples the algorithm implementation from the client code. The client interacts with the context, which delegates the algorithm to the selected strategy. This separation improves code clarity and reduces dependencies.
8. **Facilitates Better Design Patterns:** The Strategy pattern is often used in conjunction with other design patterns, such as the Factory Method pattern or the Singleton pattern, to create more sophisticated and flexible designs.

Disadvantage of Strategy Method Design Pattern in Java:

1. **Increased Number of Classes:** Implementing the Strategy pattern often involves creating a separate class for each strategy. In situations where there are numerous strategies, this can lead to an increased number of classes, potentially making the codebase more complex.
2. **Potential for Overhead:** The use of multiple strategy classes can introduce a level of indirection, which might lead to some performance overhead. The client has to instantiate and manage strategy objects, and there could be additional function call overhead when delegating to different strategies.
3. **Complexity for Small Projects:** For small projects or situations where the number of algorithms is limited and unlikely to change, applying the Strategy pattern might introduce unnecessary complexity. In such cases, a simpler solution could be more appropriate.
4. **Potential for Misuse:** If not applied judiciously, the Strategy pattern can be misused, leading to an overly complex and convoluted design. It's essential to assess whether the

flexibility provided by the pattern is genuinely needed for the specific requirements of the system.

5. **Learning Curve for Developers:** Developers who are new to the codebase might find it challenging to understand the relationships between the context and the various strategy classes. This can be mitigated with good documentation and code comments.
6. **Inherent Complexity of Algorithms:** The Strategy pattern doesn't inherently simplify the complexity of individual algorithms. If the algorithms themselves are complex, the benefits of encapsulation and interchangeability might be outweighed by the inherent intricacy of the strategies.
7. **Potential for Code Duplication:** If there are common elements across different strategies, there might be a risk of code duplication. Careful design and consideration of shared components are necessary to avoid redundancy.
8. **Difficulty in Choosing Appropriate Strategies:** In some cases, determining the most suitable strategy at runtime can be a non-trivial task. The logic for choosing strategies might become complex, especially in situations where multiple factors influence the decision.

OBSERVER DESIGN PATTERN:

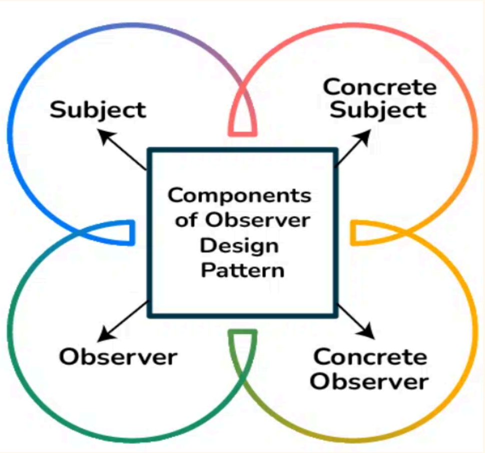
Observer Design Pattern - GeeksforGeeks

The Observer Design Pattern is a behavioral design pattern that defines a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically. It is also known as the Publish-Subscribe pattern. It primarily deals with the interaction and communication between objects, specifically focusing on how objects behave in response to changes in the state of other objects.

- The pattern is concerned with defining a mechanism for a group of objects to interact based on changes in the state of one object (the subject). The observers' behavior is triggered by changes in the subject's state.
- It encapsulates the behavior of the dependent objects (observers) and allows for a clean separation between the subject and its observers. This separation promotes a more modular and maintainable design.
- The pattern promotes loose coupling between the subject and its observers. The subject doesn't need to know the concrete classes of its observers, and observers can be added or removed without affecting the subject.
- The primary mechanism in the Observer Pattern is the notification of observers when a change occurs. This notification mechanism facilitates the dynamic and coordinated behavior of multiple objects in response to changes in the subject.

Components of the Observer Design Pattern:

Subject: The subject maintains a list of observers (subscribers or listeners). It Provides methods to register and unregister observers dynamically and defines a method to notify observers of changes in its state.



Observer: Observer defines an interface with an update method that concrete observers must implement and ensures a common or consistent way for concrete observers to receive updates from the subject. Concrete observers implement this interface, allowing them to react to changes in the subject's state.

ConcreteSubject: ConcreteSubjects are specific implementations of the subject. They hold the actual state or data that observers want to track. When this state changes, concrete subjects notify their observers. For instance, if a weather station is the subject, specific weather stations in different locations would be concrete subjects.

ConcreteObserver: Concrete Observer implements the observer interface. They register with a concrete subject and react when notified of a state change. When the subject's state changes, the concrete observer's update() method is invoked, allowing it to take appropriate actions. In a practical example, a weather app on your smartphone is a concrete observer that reacts to changes from a weather station.

When to use the Observer Design Pattern?

1. **One-to-Many Dependence:** Use the Observer pattern when there is a one-to-many relationship between objects, and changes in one object should notify multiple dependent objects. This is particularly useful when changes in one object need to propagate to several other objects without making them tightly coupled.
2. **Decoupling:** Use the Observer pattern to achieve loose coupling between objects. This allows the subject (publisher) and observers (subscribers) to interact without being aware of each other's specific details. It promotes a flexible and maintainable system.
3. **Change Propagation:** When changes in the state of one object should automatically trigger updates in other objects, the Observer pattern is beneficial. This helps ensure that all dependent objects are informed and can respond accordingly to changes in the subject.
4. **Dynamic Composition:** If you need to support dynamic composition of objects with runtime registration and deregistration of observers, the Observer pattern is suitable. New observers can be added or existing ones removed without modifying the subject.
5. **Event Handling:** The Observer pattern is often used in event handling systems. When events occur in a system, observers (listeners) can react to those events without requiring the source of the events to have explicit knowledge of the observers.

When not to use the Observer Design Pattern?

1. **Performance Overhead:** In scenarios where performance is critical and there is a concern about the overhead of managing and notifying multiple observers, the Observer pattern might not be the best choice. It adds some runtime overhead due to maintaining the list of observers and notifying them.
2. **Complexity for Simple Scenarios:** For simple scenarios where there are only a few objects that need to be notified of changes, using the Observer pattern might introduce unnecessary complexity. In such cases, a direct approach might be more straightforward.
3. **Unintended Broadcasts:** If there's a risk of unintentionally broadcasting changes to a large number of observers and you need more control over which observers should be notified, consider alternative patterns that provide more fine-grained control.
4. **Overuse in Small Systems:** In small applications where the benefits of decoupling and dynamic composition are not crucial, using the Observer pattern might be overkill. Simpler mechanisms or direct communication between objects might be more appropriate.

BUILDER DESIGN PATTERN:

The Builder Design Pattern is a creational design pattern that is used to construct complex objects step by step. It allows the construction of a product in a step-by-step fashion, where the construction process can vary based on the type of product being built. It separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

Components of the Builder Design Pattern:

Product: The Product is the complex object that the Builder pattern is responsible for constructing.

- It may consist of multiple components or parts, and its structure can vary based on the implementation.
- The Product is typically a class with attributes representing the different parts that the Builder constructs.

Builder: The Builder is an interface or an abstract class that declares the construction steps for building a complex object.

- It typically includes methods for constructing individual parts of the product.
- By defining an interface, the Builder allows for the creation of different concrete builders that can produce variations of the product.

ConcreteBuilder: ConcreteBuilder classes implement the Builder interface, providing specific implementations for building each part of the product.

- Each ConcreteBuilder is tailored to create a specific variation of the product.
- It keeps track of the product being constructed and provides methods for setting or constructing each part.

Director: The Director is responsible for managing the construction process of the complex object.

- It collaborates with a Builder, but it doesn't know the specific details about how each part of the object is constructed.
- It provides a high-level interface for constructing the product and managing the steps needed to create the complex object.

Client: The Client is the code that initiates the construction of the complex object.

- It creates a Builder object and passes it to the Director to initiate the construction process.
- The Client may retrieve the final product from the Builder after construction is complete.

When to use Builder Design Pattern?

The Builder design pattern is used when you need to create complex objects with a large number of optional components or configuration parameters. This pattern is particularly useful when an object needs to be constructed step by step, some of the scenarios where the Builder design pattern is beneficial are:

1. **Complex Object Construction:** When you have an object with many optional components or configurations and you want to provide a clear separation between the construction process and the actual representation of the object.
2. **Step-by-Step Construction:** When the construction of an object involves a step-by-step process where different configurations or options need to be set at different stages.
3. **Avoiding constructors with multiple parameters:** When the number of parameters in a constructor becomes too large, and using telescoping constructors (constructors with multiple parameters) becomes unwieldy and error-prone.
4. **Immutable Objects:** When you want to create immutable objects, and the Builder pattern allows you to construct the object gradually before making it immutable.
5. **Configurable Object Creation:** When you need to create objects with different configurations or variations, and you want a more flexible and readable way to specify these configurations.
6. **Common Interface for Multiple Representations:** When you want to provide a common interface for constructing different representations of an object.

When not to use Builder Design Pattern?

While the Builder design pattern is beneficial in many scenarios, there are situations where it might be unnecessary. Here are some cases when you might want to reconsider using the Builder pattern:

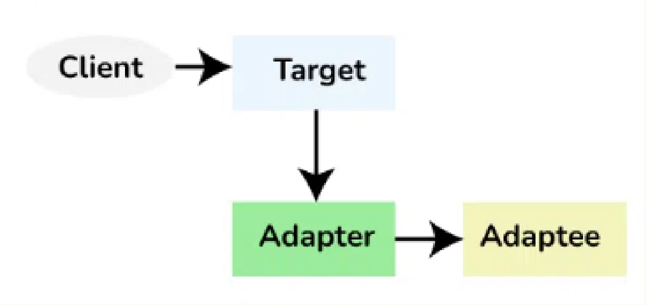
1. **Simple Object Construction:** If the object you are constructing has only a few simple parameters or configurations, and the construction process is straightforward, using a

builder might be overkill. In such cases, a simple constructor or static factory method might be more appropriate.

2. **Performance Concerns:** In performance-critical applications, the additional overhead introduced by the Builder pattern might be a concern. The extra method calls and object creations involved in the builder process could impact performance, especially if the object construction is frequent and time-sensitive.
3. **Immutable Objects with Final Fields:** If you are working with a language that supports immutable objects with final fields (e.g., Java's final keyword), and the object's structure is relatively simple, you might prefer using constructors with parameters or static factory methods.
4. **Increased Code Complexity:** Introducing a builder class for every complex object can lead to an increase in code complexity. If the object being constructed is simple and doesn't benefit significantly from a step-by-step construction process, using a builder might add unnecessary complexity to the codebase.
5. **Tight Coupling with Product:** If the builder is tightly coupled with the product it constructs, and changes to the product require corresponding modifications to the builder, it might reduce the flexibility and maintainability of the code.

ADAPTER DESIGN PATTERN:

The Adapter design pattern is a structural pattern that allows the interface of an existing class to be used as another interface. It acts as a bridge between two incompatible interfaces, making them work together. This pattern involves a single class, known as the adapter, which is responsible for joining functionalities of independent or incompatible interfaces.



Components of Adapter Design Pattern:

- **Target Interface:**
    - Description: Defines the interface expected by the client. It represents the set of operations that the client code can use.
    - Role: It's the common interface that the client code interacts with.
- **Adaptee:**
    - Description: The existing class or system with an incompatible interface that needs to be integrated into the new system.
    - Role: It's the class or system that the client code cannot directly use due to interface mismatches.
- **Adapter:**
    - Description: A class that implements the target interface and internally uses an instance of the adaptee to make it compatible with the target interface.
    - Role: It acts as a bridge, adapting the interface of the adaptee to match the target interface.
- **Client:**
    - Description: The code that uses the target interface to interact with objects. It remains unaware of the specific implementation details of the adaptee and the adapter.
    - Role: It's the code that benefits from the integration of the adaptee into the system through the adapter.

How does the Adapter Design Pattern work?

Client Request: The client initiates a request by calling a method on the adapter using the target interface.
Adapter Translation: The adapter translates or maps the client's request into a form that the adaptee understands, using the adaptee's interface.
Adaptee Execution: The adaptee performs the actual work based on the translated request from the adapter.
Result to Client: The client receives the results of the call, remaining unaware of the adapter's presence or the specific details of the adaptee.

Why do we need Adapter Design Pattern?

- **Integration of Existing Code:**
  - Scenario: When you have existing code or components with interfaces that are incompatible with the interfaces expected by new code or systems.
  - Need: The Adapter pattern allows you to integrate existing components seamlessly into new systems without modifying their original code.
- **Reuse of Existing Functionality:**
  - Scenario: When you want to reuse classes or components that provide valuable functionality but don't conform to the desired interface.
  - Need: The Adapter pattern enables you to reuse existing code by creating an adapter that makes it compatible with the interfaces expected by new code.
- **Interoperability:**
  - Scenario: When you need to make different systems or components work together, especially when they have different interfaces.
  - Need: The Adapter pattern acts as a bridge, allowing systems with incompatible interfaces to collaborate effectively.
- **Client-Server Communication:**
  - Scenario: When building client-server applications, and the client expects a specific interface while the server provides a different one.
  - Need: Adapters help in translating requests and responses between client and server, ensuring smooth communication despite interface differences.
- **Third-Party Library Integration:**
  - Scenario: When incorporating third-party libraries or APIs into a project, and their interfaces do not match the rest of the system.
  - Need: Adapters make it possible to use external components by providing a compatible interface for the rest of the application.

When not to use Adapter Design Pattern?

- **When Interfaces Are Stable:**
  - Scenario: If the interfaces of the existing system and the new system are stable and not expected to change frequently.
  - Reason: Adapters are most beneficial when dealing with evolving or incompatible interfaces. If the interfaces are stable, the overhead of maintaining adapters might outweigh the benefits.
- **When Direct Modification Is Feasible:**
  - Scenario: If you have control over the source code of the existing system, and it's feasible to directly modify its interface to match the target interface.
  - Reason: If you can modify the existing code, direct adaptation of interfaces might be a simpler and more straightforward solution than introducing adapters.
- **When Performance is Critical:**
  - Scenario: In performance-critical applications where the overhead introduced by the Adapter pattern is not acceptable.

    - Reason: Adapters may introduce a level of indirection and abstraction, which could have a minor impact on performance. In situations where every bit of performance matters, the Adapter pattern might not be the best choice.
- **When Multiple Adapters Are Required:**
    - Scenario: If a system requires numerous adapters for various components, and the complexity of managing these adapters becomes overwhelming.
    - Reason: Managing a large number of adapters might lead to increased complexity and maintenance challenges. In such cases, reconsider the overall design or explore alternatives.
- **When Adapters Introduce Ambiguity:**
    - Scenario: When introducing adapters leads to ambiguity or confusion in the overall system architecture.
    - Reason: If the presence of adapters makes the system design less clear or harder to understand, it may be worthwhile to explore alternative solutions that offer a clearer design.

DECORATOR DESIGN PATTERN:

The Decorator Design Pattern is a structural design pattern that allows behavior to be added to individual objects dynamically, without affecting the behavior of other objects from the same class. It is used to extend or modify the functionality of objects at runtime by wrapping them with additional behavior in a flexible and reusable way.

- It uses inheritance or composition to extend the behavior of an object but this is done at compile time
- Decorator patterns allow a user to add new functionality to an existing object without altering its structure. So, there is no change to the original class.
- The decorator design pattern is a structural pattern, which provides a wrapper to the existing class.
- The decorator design pattern uses abstract classes or interfaces with the composition to implement the wrapper.
- Decorator design patterns create decorator classes, which wrap the original class and supply additional functionality by keeping the class methods' signature unchanged.
- Decorator design patterns are most frequently used for applying single responsibility principles since we divide the functionality into classes with unique areas of concern.
- The decorator design pattern is structurally almost like the chain of responsibility pattern.

Components of the Decorator Design Pattern:

- **Component:** Defines the interface for objects that can have responsibilities added to them dynamically.
- **Concrete Component:** Implements the Component interface and represents the original object to which additional functionality can be added.
- **Decorator:** Abstract class that implements the Component interface and maintains a reference to a Component object and it has a HAS-A relationship with the component interface. It acts as a base class for concrete decorators.
- **Concrete Decorator:** Extends the Decorator class and adds additional behavior or responsibilities to the component.

Procedure:

- Create an interface.
- Create concrete classes implementing the same interface.
- Create an abstract decorator class implementing the above same interface.
- Create a concrete decorator class extending the above abstract decorator class.
- Now use the concrete decorator class created above to decorate interface objects.

- Lastly, verify the output

Benefits of the Decorator Pattern:

- **Flexibility:** Allows for adding or removing responsibilities dynamically at runtime.
- **Open/Closed Principle:** Promotes open/closed principle as it allows extension without modification of existing code.
- **Composition over Inheritance:** Provides an alternative to subclassing for extending functionality.

Drawbacks:

- **Complexity:** Can lead to a large number of small classes if used excessively, which may increase code complexity.
- **Object Proliferation:** The use of decorators can lead to a large number of objects being created, impacting memory consumption.

When to Use:

- Use the Decorator pattern when you need to add functionality to objects dynamically without altering their structure.
- Use it when subclassing is impractical or leads to a complex class hierarchy.
- Use it to maintain the single responsibility principle by separating concerns into small, focused classes.

CHAIN OF RESPONSIBILITY DESIGN PATTERN:

The Chain of Responsibility Design Pattern is a behavioral design pattern that allows a request to be processed by a chain of handlers. Each handler in the chain has the ability to either handle the request or pass it on to the next handler in the chain. It decouples the sender of a request from its receiver, providing a way to process requests dynamically without requiring the sender to know which object will ultimately handle the request.

Characteristics of the Chain of Responsibility Design Pattern:

- **Loose Coupling:** The pattern promotes loose coupling between the sender and receiver of a request, as the sender doesn’t need to know which object will handle the request and the receiver doesn’t need to know the structure of the chain.
- **Dynamic Chain:** The chain can be modified dynamically at runtime, allowing for flexibility in adding or removing handlers without affecting the client code.
- **Single Responsibility Principle:** Each handler in the chain has a single responsibility, either handling the request or passing it to the next handler, which helps in maintaining a clean and modular design.
- **Sequential Order:** Requests are processed sequentially along the chain, ensuring that each request is handled in a predefined order.
- **Fallback Mechanism:** The chain can include a mechanism to handle requests that are not handled by any handler in the chain, providing a fallback or default behavior.
- **Variants:** The pattern has variants like a linear chain, where each handler has a single successor, or a tree-like structure, where a handler can have multiple successors, allowing for more complex processing logic.
- **Enhanced Flexibility:** The pattern allows for enhanced flexibility in handling requests, as the chain can be configured or modified to suit different requirements without changing the client code.

Components of the Chain of Responsibility Design Pattern:

1. **Handler Interface or Abstract Class:** This is the base class that defines the interface for handling requests and, in many cases, for chaining to the next handler in the sequence.

2. **Concrete Handlers:** These are the classes that implement how the requests are going to be handled. They can handle the request or pass it to the next handler in the chain if it is unable to handle that request.
3. **Client:** The request is sent by the client, who then forwards it to the chain's first handler. Which handler will finally handle the request is unknown to the client.

Advantages of the Chain of Responsibility Design Pattern:

1. **Decoupling of Objects:** The pattern enables sending a request to a series of possible recipients without having to worry about which object will handle it in the end. This lessens the reliance between items.
2. **Flexibility and Extensibility:** New handlers can be easily added or existing ones can be modified without affecting the client code. This promotes flexibility and extensibility within the system.
3. **Dynamic Order of Handling:** The sequence and order of handling requests can be changed dynamically during runtime, which allows adjustment of the processing logic as per the requirements.
4. **Simplified Object Interactions:** It simplifies the interaction between the sender and receiver objects, as the sender does not need to know about the processing logic.
5. **Enhanced Maintainability:** Each handler performs a specific type of processing, which makes maintaining and modifying the individual components easier without impacting the overall system.

Disadvantages of the Chain of Responsibility Design Pattern:

1. **Possible Unhandled Requests**: The chain should be implemented correctly otherwise there is a chance that some requests might not get handled at all, which leads to unexpected behavior in the application.
2. **Performance Overhead:** The request will go through several handlers in the chain if it is lengthy and complicated, which could cause performance overhead. The processing logic of each handler has an effect on the system's overall performance.
3. **Complexity in Debugging:** The fact that the chain has several handlers can make debugging more difficult. Tracking the progression of a request and determining which handler is in charge of handling it can be difficult.
4. **Runtime Configuration Overhead:** It may become more difficult to manage and maintain the chain of responsibility if the chain is dynamically modified at runtime.

When to Use:

- Use the Chain of Responsibility pattern when you have a series of handlers that need to process a request in a specific order.
- Use it when you want to decouple the sender and receiver of a request and allow multiple objects to handle the request without the sender knowing which object will handle it.

PROTOTYPE DESIGN PATTERN:

The Prototype Design Pattern is a creational design pattern used to create new objects by cloning an existing object, known as the prototype. Instead of creating objects using a constructor, the Prototype pattern allows us to create new objects by copying an existing object, thereby avoiding the need for subclassing. Prototype allows us to hide the complexity of making new instances from the client.

- The newly copied object may change the same properties only if required. This approach saves costly resources and time, especially when object creation is a heavy process.

- The prototype pattern is a creational design pattern. Prototype patterns are required when object creation is a time-consuming, and costly operation, so we create objects with the existing object itself.
- One of the best available ways to create an object from existing objects is the clone() method. Clone is the simplest approach to implementing a prototype pattern. However, it is your call to decide how to copy existing objects based on your business model.

Components of Prototype Design Pattern:

1. **Prototype Interface or Abstract Class:** The Prototype Interface or Abstract Class declares the method(s) for cloning an object. It defines the common interface that concrete prototypes must implement, ensuring that all prototypes can be cloned in a consistent manner.
    a. The main role is to provide a blueprint for creating new objects by specifying the cloning contract.
    b. It declares the clone method, which concrete prototypes implement to produce copies of themselves.
2. **Concrete Prototype:** The Concrete Prototype is a class that implements the prototype interface or extends the abstract class. It's the class representing a specific type of object that you want to clone.
    a. It defines the details of how the cloning process should be carried out for instances of that class.
    b. Implements the clone method declared in the prototype interface, providing the cloning logic specific to the class.
3. **Client:** The Client is the code or module that requests the creation of new objects by interacting with the prototype. It initiates the cloning process without being aware of the concrete classes involved.
4. **Clone Method:** The Clone Method is declared in the prototype interface or abstract class. It specifies how an object should be copied or cloned. Concrete prototypes implement this method to define their unique cloning behavior. It Describes how the object's internal state should be duplicated to create a new, independent instance.

When to use the Prototype Design Pattern?

- **Creating Objects is Costly:**
    - Use the Prototype pattern when creating objects is more expensive or complex than copying existing ones.
    - If object creation involves significant resources, such as database or network calls, and you have a similar object available, cloning can be more efficient.
- **Variations of Objects:**
    - Use the Prototype pattern when your system needs to support a variety of objects with slight variations.
    - Instead of creating multiple classes for each variation, you can create prototypes and clone them with modifications.
- **Dynamic Configuration:**
    - Use the Prototype pattern when your system requires dynamic configuration and you want to create objects with configurations at runtime. You can prototype a base configuration and clone it, adjusting the properties as needed.
- **Reducing Initialization Overhead:**
    - Use the Prototype pattern when you want to reduce the overhead of initializing an object. Creating a clone can be faster than creating an object from scratch, especially when the initialization process is resource-intensive.

When not to use the Prototype Design Pattern?

- **Unique Object Instances:**
  - Avoid using the Prototype pattern when your application predominantly deals with unique object instances, and the overhead of implementing the pattern outweighs its benefits.
- **Simple Object Creation:**
  - If object creation is simple and does not involve significant resource consumption, and there are no variations of objects, using the Prototype pattern might be unnecessary complexity.
- **Immutable Objects:**
  - If your objects are immutable (unchangeable) and do not need variations, the benefits of cloning may not be significant. Immutable objects are often safely shared without the need for cloning.
- **Clear Object Creation Process:**
  - If your system has a clear and straightforward object creation process that is easy to understand and manage, introducing the Prototype pattern may add unnecessary complexity.
- **Limited Object Variations:**
  - If there are only a few variations of objects, and creating subclasses or instances with specific configurations is manageable, the Prototype pattern might be overkill.

COMMAND DESIGN PATTERN:

The Command Design Pattern is a behavioral design pattern that encapsulates a request as an object, thereby allowing parameterization of clients with queues, requests, and operations. It enables decoupling between sender and receiver of a request, thus promoting loose coupling in a system. This pattern is commonly used to implement undo functionality, transaction management, and in scenarios where you need to support different types of requests and operations.

- The Command Pattern turns a request into a stand-alone object , allowing for the separation of sender and receiver.
- Commands can be parameterized, meaning you can create different commands with different parameters without changing the invoker(responsible for initiating command execution).
- It decouples the sender (client or invoker) from the receiver (object performing the operation), providing flexibility and extensibility.
- The pattern supports undoable(action or a series of actions that can be reversed or undone in a system) operations by storing the state or reverse commands.

Components of the Command Design Pattern:

1. **Command Interface:** The Command Interface is like a rulebook that all command classes follow. It declares a common method, execute(), ensuring that every concrete command knows how to perform its specific action. It sets the standard for all commands, making it easier for the remote control to manage and execute diverse operations without needing to know the details of each command.
2. **Concrete Command Classes:** Concrete Command Classes are the specific commands, like turning on a TV or adjusting the stereo volume. Each class encapsulates the details of a particular action. These classes act as executable instructions that the remote control can trigger without worrying about the nitty-gritty details of how each command accomplishes its task.
3. **Invoker (Remote Control):** The Invoker, often a remote control, is the one responsible for initiating command execution. It holds a reference to a command but doesn't delve into the specifics of how each command works. It's like a button that, when pressed, makes things

happen. The remote control's role is to coordinate and execute commands without getting involved in the complexities of individual actions.

4. **Receiver (Devices):** The Receiver is the device that knows how to perform the actual operation associated with a command. It could be a TV, stereo, or any other device. Receivers understand the specific tasks mentioned in commands. If a command says, "turn on," the Receiver (device) knows precisely how to execute that action. The Receiver-Command relationship separates responsibilities, making it easy to add new devices or commands without messing with existing functionality.

When to use the Command Design Pattern?

- **Decoupling is Needed:**
    - Use the Command Pattern when you want to decouple the sender (requester) of a request from the object that performs the request.
    - This helps in making your code more flexible and extensible.
- **Undo/Redo Functionality is Required:**
    - If you need to support undo and redo operations in your application, the Command Pattern is a good fit.
    - Each command can encapsulate an operation and its inverse, making it easy to undo or redo actions.
- **Support for Queues and Logging:**
    - If you want to maintain a history of commands, log them, or even put them in a queue for execution, the Command Pattern provides a structured way to achieve this.
- **Dynamic Configuration:**
    - When you need the ability to dynamically configure and assemble commands at runtime, the Command Pattern allows for flexible composition of commands.

When not to use the Command Design Pattern?

- **Simple Operations:**
    - For very simple operations or one-off tasks, introducing the Command Pattern might be overkill.
    - It's beneficial when you expect your operations to become more complex or when you need to support undo/redo.
- **Tight Coupling is Acceptable:**
    - If the sender and receiver of a request are tightly coupled and changes in one do not affect the other, using the Command Pattern might introduce unnecessary complexity.
- **Overhead is a Concern:**
    - In scenarios where performance and low overhead are critical factors, introducing the Command Pattern might add some level of indirection and, in turn, impact performance.
- **Limited Use of Undo/Redo:**
    - If your application does not require undo/redo functionality and you do not anticipate needing to support such features in the future, the Command Pattern might be unnecessary complexity.

TEMPLATE DESIGN PATTERN:

The Template Method Pattern is a behavioral design pattern that defines the skeleton of an algorithm in a superclass but lets subclasses override specific steps of the algorithm without changing its structure. It promotes code reuse by encapsulating the common algorithmic structure in the superclass while allowing subclasses to provide concrete implementations for certain steps, thus enabling customization and flexibility.

- The overall structure and sequence of the algorithm are preserved by the parent class.
- Template means Preset format like HTML templates which have a fixed preset format. Similarly in the template method pattern, we have a preset structure method called template method which consists of steps.
- These steps can be an abstract method that will be implemented by its subclasses.
- This is one of the easiest to understand and implement. This design pattern is used popularly in framework development and helps to avoid code duplication.

Components of Template Method Design Pattern:

- **Abstract Class (or Interface):** This is the superclass that defines the template method. It provides a skeleton for the algorithm, where certain steps are defined but others are left abstract or defined as hooks that subclasses can override. It may also include concrete methods that are common to all subclasses and are used within the template method.
- **Template Method:** This is the method within the abstract class that defines the overall algorithm structure by calling various steps in a specific order. It's often declared as final to prevent subclasses from changing the algorithm's structure. The template method usually consists of a series of method calls (either abstract or concrete) that make up the algorithm's steps.
- **Abstract (or Hook) Methods:** These are methods declared within the abstract class but not implemented. They serve as placeholders for steps in the algorithm that should be implemented by subclasses. Subclasses must provide concrete implementations for these methods to complete the algorithm.
- **Concrete Subclasses:** These are the subclasses that extend the abstract class and provide concrete implementations for the abstract methods defined in the superclass. Each subclass can override certain steps of the algorithm to customize the behavior without changing the overall structure.

When to use the Template Method Design Pattern?

- **Common Algorithm with Variations:** When you have an algorithm with a common structure but with some varying steps or implementations, the Template Method pattern helps to encapsulate the common steps in a superclass while allowing subclasses to override specific steps.
- **Code Reusability:** If you have similar tasks or processes that need to be performed in different contexts, the Template Method pattern promotes code reuse by defining the common steps in one place.
- **Enforcing Structure:** It's beneficial when you want to enforce a specific structure or sequence of steps in an algorithm while allowing for flexibility in certain parts.
- **Reducing Duplication:** By centralizing common behavior in the abstract class and avoiding duplication of code in subclasses, the Template Method pattern helps in maintaining a clean and organized codebase.

When not to use the Template Method Design Pattern?

- **When Algorithms are Highly Variable:** If the algorithms you're working with vary greatly in their structure and steps, and there's minimal commonality between them, using the Template Method pattern might not be appropriate as it may lead to excessive complexity or unnecessary abstraction.
- **Tight Coupling Between Steps:** If there's tight coupling between the steps of the algorithm, such that changes in one step necessitate changes in other steps, the Template Method pattern may not provide sufficient flexibility.
- **Inflexibility with Runtime Changes:** If you anticipate frequent changes in the algorithm structure or steps at runtime, using the Template Method pattern might not be the best choice, as it relies on predefined structure and behavior.

- **Overhead in Abstraction:** If the cost of abstraction and inheritance outweighs the benefits of code reuse and structure enforcement, it's better to avoid using the Template Method pattern and opt for simpler solutions.

MEDIATOR DESIGN PATTERN:

The Mediator design pattern is a behavioral design pattern that promotes loose coupling by encapsulating the way objects interact with each other. It defines an object (the mediator) that centralizes communication between multiple objects (colleagues) instead of having them communicate directly with each other. This reduces the dependencies between the objects and makes them easier to maintain and extend.

- If the objects interact with each other directly, the system components are tightly coupled with each other making higher maintainability cost and not hard to extend.
- The mediator pattern focuses on providing a mediator between objects for communication and helps in implementing loose coupling between objects.

Components of the Mediator Design Pattern:

- **Mediator:** The Mediator interface defines the communication contract, specifying methods that concrete mediators should implement to facilitate interactions among colleagues.. It encapsulates the logic for coordinating and managing the interactions between these objects, promoting loose coupling and centralizing control over their communication.
- **Colleague:** Colleague classes are the components or objects that interact with each other. They communicate through the Mediator, and each colleague class is only aware of the mediator, not the other colleagues. This isolation ensures that changes in one colleague do not directly affect others.
- **Concrete Mediator:** Concrete Mediator is a specific implementation of the Mediator interface. It coordinates the communication between concrete colleague objects, handling their interactions and ensuring a well-organized collaboration while keeping them decoupled.
- **Concrete colleague:** Concrete Colleague classes are the specific implementations of the Colleague interface. They rely on the Mediator to communicate with other colleagues, avoiding direct dependencies and promoting a more flexible and maintainable system architecture.

When to use the Mediator Design Pattern?

- **Complex Communication:** Your system involves a set of objects that need to communicate with each other in a complex manner, and you want to avoid direct dependencies between them.
- **Loose Coupling:** You want to promote loose coupling between objects, allowing them to interact without knowing the details of each other's implementations.
- **Centralized Control:** You need a centralized mechanism to coordinate and control the interactions between objects, ensuring a more organized and maintainable system.
- **Changes in Behavior:** You anticipate changes in the behavior of components, and you want to encapsulate these changes within the mediator, preventing widespread modifications.
- **Enhanced Reusability:** You want to reuse individual components in different contexts without altering their internal logic or communication patterns.

When not to use the Mediator Design Pattern?

- **Simple Interactions:** The interactions between components are straightforward, and introducing a mediator would add unnecessary complexity.

- **Single Responsibility Principle (SRP):** Each component has a single responsibility, and introducing a mediator might violate the Single Responsibility Principle, leading to less maintainable code.
- **Performance Concerns:** Introducing a mediator could introduce a performance overhead, especially in situations where direct communication between components is more efficient.
- **Small Scale Applications:** In small-scale applications with a limited number of components, the overhead of implementing a mediator might outweigh its benefits.
- **Over-Engineering:** Avoid using the Mediator pattern if it seems like an over-engineered solution for the specific requirements of your system. Always consider the trade-offs and the specific needs of your application.

MEMENTO DESIGN PATTERN:

The Memento design pattern is a behavioral pattern that is used to capture and restore an object's internal state without violating encapsulation. It allows you to save and restore the state of an object to a previous state, providing the ability to undo or roll back changes made to the object.

- As the application progresses, we may want to save checkpoints in our application and restore them to those checkpoints later.
- The intent of the Memento Design pattern is, without violating encapsulation, to capture and externalize an object's internal state so that the object can be restored to this state later.

Components of Memento Design Pattern:

- **Originator:** This component is responsible for creating and managing the state of an object. It has methods to set and get the object's state, and it can create Memento objects to store its state. The Originator communicates directly with the Memento to create snapshots of its state and to restore its state from a snapshot.
- **Memento:** The Memento is an object that stores the state of the Originator at a particular point in time. It only provides a way to retrieve the state, without allowing direct modification. This ensures that the state remains
- **Caretaker:** The Caretaker is responsible for keeping track of Memento objects. It doesn't know the details of the state stored in the Memento but can request Mementos from the Originator to save or restore the object's state.
- **Client:** Typically represented as the part of the application or system that interacts with the Originator and Caretaker to achieve specific functionality. The client initiates requests to save or restore the state of the Originator through the Caretaker.

Communication between the components:

- **Client:** The client initiates the process by requesting the Originator to perform some operation that may modify its state or require the state to be saved. For example, the client might trigger an action like "save state" or "restore state."
- **Originator:** Upon receiving a request from the client, the Originator either creates a Memento to save its current state (if the request is to save state) or retrieves a Memento to restore its previous state (if the request is to restore state).
  - **If the request is to save state:**
    - The Originator creates a Memento object that captures its current state.
    - It returns the Memento to the client or Caretaker for storage.
  - **If the request is to restore state:**
    - The Originator retrieves the desired Memento containing the state it wants to restore.
    - It restores its state using the state stored in the Memento.

- **Caretaker:** The Caretaker acts as an intermediary between the client and the Originator, managing the Memento objects.
    - **If the client requests to save state:**
        - The Caretaker receives the Memento from the Originator.
        - It stores the Memento for future use.
    - **If the client requests to restore state:**
        - The Caretaker retrieves the appropriate Memento from its storage.
        - It provides the Memento to the Originator for state restoration.
        - The caretaker calls the createMemento() method on the originator asking the originator to pass it a memento object.
        - At this point the originator creates a memento object saving its internal state and passes the memento to the caretaker.
        - The caretaker maintains the memento object and performs the operation. In case of the need to undo the operation, the caretaker calls the setMemento() method on the originator passing the maintained memento object.
        - The originator would accept the memento, using it to restore its previous state.

When to use Memento Design Pattern?

- **Undo functionality:** When you need to implement an undo feature in your application that allows users to revert changes made to an object's state.
- **Snapshotting:** When you need to save the state of an object at various points in time to support features like versioning or checkpoints.
- **Transaction rollback:** When you need to rollback changes to an object's state in case of errors or exceptions, such as in database transactions.
- **Caching:** When you want to cache the state of an object to improve performance or reduce redundant computations.

When not to use Memento Design Pattern?

- **Large object state:** If the object's state is large or complex, storing and managing multiple snapshots of its state can consume a significant amount of memory and processing resources.
- **Frequent state changes:** If the object's state changes frequently and unpredictably, storing and managing snapshots of its state may become impractical or inefficient.
- **Immutable objects:** If the object's state is immutable or easily reconstructible, there may be little benefit in using the Memento pattern to capture and restore its state.
- **Overhead:** Introducing the Memento pattern can add complexity to the codebase, especially if the application does not require features like undo functionality or state rollback.

STATE DESIGN PATTERN:

The State design pattern is a behavioral design pattern that allows an object to change its behavior when its internal state changes. This pattern is particularly useful when the behavior of an object depends on its state and there are multiple states with different behaviors, and the state can change during the object's lifecycle. By encapsulating each state into a separate class and delegating the state-specific behavior to these classes, the State pattern promotes encapsulation, flexibility, and extensibility.

Components of State Design Pattern:

- **Context:** The Context is the class that contains the object whose behavior changes based on its internal state. It maintains a reference to the current state object that represents the current state of the Context. The Context provides an interface for clients to interact with and typically delegates state-specific behavior to the current state object.

- **State Interface or Base Class:** The State interface or base class defines a common interface for all concrete state classes. This interface typically declares methods that represent the state-specific behavior that the Context can exhibit. It allows the Context to interact with state objects without knowing their concrete types.
- **Concrete States:** Concrete state classes implement the State interface or extend the base class. Each concrete state class encapsulates the behavior associated with a specific state of the Context. These classes define how the Context behaves when it is in their respective states.

Communication between the components:

In the State design pattern, the communication between the components typically follows these steps:

- **Step1: Client Interaction:** The client interacts with the Context object, either directly or indirectly, by invoking methods on it.
- **Step2 :Behavior Delegation:** When the client triggers an action or requests a behavior from the Context, the Context delegates the responsibility to its current State object.
- **Step3: State-specific Behavior Execution:** The current State object receives the delegated request and executes the behavior associated with its particular state.
- **Step4: Possible State Transition:**
- Depending on the logic implemented within the State object or controlled by the Context, a state transition may occur.
- **Step5: Update of Current State:**
- If a state transition occurs, the Context updates its reference to the new State object, reflecting the change in its internal state.
- **Step6: Continued Interaction:**
- The client continues to interact with the Context as needed, and the process repeats, with behavior delegation to the appropriate State object based on the current state of the Context.

When to use the State Design Pattern?

The State design pattern is beneficial when you encounter situations with objects whose behavior changes dynamically based on their internal state. Here are some key indicators:

- **Multiple states with distinct behaviors:** If your object exists in several states (e.g., On/Off, Open/Closed, Started/Stopped), and each state dictates unique behaviors, the State pattern can encapsulate this logic effectively.
- **Complex conditional logic:** When conditional statements (if-else or switch-case) become extensive and complex within your object, the State pattern helps organize and separate state-specific behavior into individual classes, enhancing readability and maintainability.
- **Frequent state changes:** If your object transitions between states frequently, the State pattern provides a clear mechanism for managing these transitions and their associated actions.
- **Adding new states easily:** If you anticipate adding new states in the future, the State pattern facilitates this by allowing you to create new state classes without affecting existing ones.

When not to use the State Design Pattern?

While the State pattern offers advantages, it’s not always the best solution. Here are some cases where it might be overkill:

- **Few states with simple behavior:** If your object has only a few simple states with minimal behavioral differences, the overhead of the State pattern outweighs its benefits. In such cases, simpler conditional logic within the object itself might suffice.
- **Performance-critical scenarios:** The pattern can introduce additional object creation and method calls, potentially impacting performance. If performance is paramount, a different approach might be more suitable.
- **Over-engineering simple problems:** Don't apply the pattern just for the sake of using a design pattern. If your logic is clear and maintainable without it, stick with the simpler solution.

State vs Strategy Design Pattern:

STATE PATTERN	STRATEGY PATTERN
In-State pattern, an individual state can contain the reference of Context, to implement state transitions.	But Strategies don't contain the reference of Context, where they are used.
State encapsulates the state of an Object.	While Strategy Pattern encapsulates an algorithm or strategy.
State pattern helps a class to exhibit different behaviors in a different state.	Strategy Pattern encapsulates a set of related algorithms and allows the client to use interchangeable behaviors through composition and delegation at runtime.
The state is part of the context object itself, and over time, the context object transitions from one State to another.	It can be passed as a parameter to the Object which uses them e.g. Collections.sort() accepts a Comparator, which is a strategy.
State Pattern defines the “what” and “when” part of an Object. Example: What can an object when it's in a certain state.	Strategy pattern defines the “How” part of an Object. Example: How a Sorting object sorts data.
Order of State transition is well-defined in State pattern.	There is no such requirement for a Strategy pattern. The client is free to choose any Strategy implementation of his choice.
Change in State can be done by Context or State object itself.	Change in Strategy is done by the Client.
Some common examples of Strategy Patterns are encapsulating algorithms e.g. sorting algorithms, encryption algorithms, or compression algorithms.	On the other hand, recognizing the use of State design patterns is pretty easy.

If you see, your code needs to use different kinds of related algorithms, then think of using a Strategy pattern.	If you need to manage state and state transition, without lots of nested conditional statements, state pattern is the pattern to use.

ITERATOR DESIGN PATTERN:

The Iterator Design Pattern is a behavioral design pattern that provides a way to access elements of an aggregate object (such as a list) sequentially without exposing its underlying representation. It allows clients to traverse the elements of a collection without needing to know the internal structure of the collection.

- The iterator pattern is a great pattern for providing navigation without exposing the structure of an object.
- Traverse a container. In Java and most current programming languages, there's the notion of a collection. List, Maps, Sets are all examples of a collection that we would want to traverse. Historically we use a loop of some sort and index into your collection to traverse it.
- Doesn't expose the underlying structure of the object we want to navigate. Navigating various structures may have different algorithms or approaches to cycle through the data.
- Decouples the data from the algorithm used to traverse it
- It is an interface-based design pattern. Whichever object you want to iterate over will provide a method to return an instance of an iterator from it.
- Follows a factory-based method pattern in the way you get an instance of the iterator.
- Each iterator is developed in such a way that it is independent of another.
- Iterators also Fail Fast. Fail Fast means that iterators can't modify the underlying object without an error being thrown.

Components of Iterator Design Pattern:

- **Iterator Interface:** Defines the interface for accessing and traversing elements of the collection. It typically includes methods like hasNext() to check if there are more elements, and next() to retrieve the next element.
- **Concrete Iterator:** Implements the iterator interface and provides the mechanism to traverse the elements of the collection. It maintains the current position within the collection.
- **Aggregate Interface:** Defines the interface for creating iterators. It may include a method like createIterator() to create an iterator object for the collection.
- **Concrete Aggregate:** Implements the aggregate interface and represents a collection of elements. It provides an implementation for creating iterators specific to its type.

When Will We Need Iterator Design Pattern?

Each and every software programming language has data structures such as lists and maps that would be used to hold a set of associated elements. In the programming language Java, users have many interfaces like List, Map, Sets as well as various implementations like HashMaps and ArrayLists.

A collection can be useful if it grants access to its components while not disclosing its inner structure. That's the job of the iterators.

So, whenever we have a number of objects and when users want a mechanism to loop through each collection piece in a specific order, we should utilize the iterator pattern to construct the solution.

It is utilized to navigate through a container as well as access its elements. Algorithms and containers are separated using the iterator pattern.

Let's suppose we want to develop a new data structure which is a collection, we can then use the iterator pattern. If someone wants to use the data structure, they can simply use the iterators you describe rather than having to create a unique iterator in their code.
The iterator pattern allow us to design a collection iterator in such a way that –

- we are able to access elements of a collection without exposing the internal structure of elements or collection itself.
- iterator supports multiple simultaneous traversals of a collection from start to end in forward, backward or both directions.
- iterator provides a uniform interface for traversing different collection types transparently.

Pros of Iterator Design Pattern

- The concept of single responsibility. By separating cumbersome traversal methods into different classes, you may clean up the client code as well as collections.
- The concept of open/closed - The programmer can introduce additional kinds of collections as well as iterators to the code which already exists without affecting anything.
- Since each iterator object has its own iteration state, you may loop over the similar collection concurrently.
- Now for the exact cause you can keep delaying an iteration and then resume it later if necessary.

Cons of Iterator Design Pattern

- If your software simply interacts with basic collections, using the Iterator Design pattern may be an overkill in that case.
- Explicitly traversing items of some specialized collections may be more competent than using an iterator design pattern.

Iterator vs Template Design Pattern:

Iterator Design Pattern	Template Design Pattern
The Iterator Pattern captures the process of iterating through a group of objects.	The template method design pattern belongs to the behavioral design pattern category. It refers to a method that is implemented as a skeleton of functions.

VISITOR DESIGN PATTERN:

The Visitor Design Pattern is a behavioral design pattern that allows adding new operations to existing classes without modifying those classes. It is useful when you have a set of related classes with different interfaces and you want to perform operations on these classes based on their types. With the help of visitor pattern, we can move the operational logic from the objects to another class. The visitor pattern consists of two parts:

- a method called Visit() which is implemented by the visitor and is called for every element in the data structure
- visitable classes providing Accept() methods that accept a visitor

Components of Iterator Design Pattern:

- **Client:** The Client class is a consumer of the classes of the visitor design pattern. It has access to the data structure objects and can instruct them to accept a Visitor to perform the appropriate processing.
- **Visitor:** This is an interface or an abstract class used to declare the visit operations for all the types of visitable classes.
- **ConcreteVisitor:** For each type of visitor all the visit methods, declared in abstract visitor, must be implemented. Each Visitor will be responsible for different operations.

- **Visitable:** This is an interface which declares the accept operation. This is the entry point which enables an object to be "visited" by the visitor object.
- **ConcreteVisitable:** These classes implement the Visitable interface or class and define the accept operation. The visitor object is passed to this object using the accept operation.

Advantages:

- If the logic of operation changes, then we need to make changes only in the visitor implementation rather than doing it in all the item classes.
- Adding a new item to the system is easy, it will require change only in visitor interface and implementation and existing item classes will not be affected.

Disadvantages:

- We should know the return type of visit() methods at the time of designing otherwise we will have to change the interface and all of its implementations.
- If there are too many implementations of visitor interface, it makes it hard to extend.

BRIDGE DESIGN PATTERN:

The Bridge Design Pattern is a structural design pattern that decouples an abstraction from its implementation, allowing them to vary independently. It involves creating two separate hierarchies: one for abstraction (interface or abstract class) and another for implementation (concrete class or interface). The bridge pattern is particularly useful when you want to avoid a permanent binding between an abstraction and its implementation, allowing both to evolve independently.

There are 2 parts in Bridge design pattern :

1. Abstraction
2. Implementation

This is a design mechanism that encapsulates an implementation class inside of an interface class.

- The bridge pattern allows the Abstraction and the Implementation to be developed independently and the client code can access only the Abstraction part without being concerned about the Implementation part.
- The abstraction is an interface or abstract class and the implementer is also an interface or abstract class.
- The abstraction contains a reference to the implementer. Children of the abstraction are referred to as refined abstractions, and children of the implementer are concrete implementers. Since we can change the reference to the implementer in the abstraction, we are able to change the abstraction's implementer at run-time. Changes to the implementer do not affect client code.
- It increases the loose coupling between class abstraction and its implementation.

Components of Iterator Design Pattern:

- **Abstraction:** core of the bridge design pattern and defines the crux. Contains a reference to the implementer.
- **Refined Abstraction:** Extends the abstraction takes the finer detail one level below. Hides the finer elements from implementers.
- **Implementer:** It defines the interface for implementation classes. This interface does not need to correspond directly to the abstraction interface and can be very different. Abstraction imp provides an implementation in terms of operations provided by the Implementer interface.
- **Concrete Implementation:** Implements the above implementer by providing the concrete implementation.

When do we need bridge design pattern?

The Bridge pattern is an application of the old advice, “prefer composition over inheritance”. It becomes handy when you must subclass different times in ways that are orthogonal with one another.

Advantages:

- Bridge pattern decouple an abstraction from its implementation so that the two can vary independently.
- It is used mainly for implementing platform independence features.
- It adds one more method level redirection to achieve the objective.
- Publish the abstraction interface in a separate inheritance hierarchy, and put the implementation in its own inheritance hierarchy.
- Use bridge pattern to run-time binding of the implementation.
- Use bridge pattern to map orthogonal class hierarchies
- Bridge is designed up-front to let the abstraction and the implementation vary independently.

COMPOSITE DESIGN PATTERN:

The Composite Design Pattern is a structural design pattern that allows you to compose objects into tree-like structures to represent part-whole hierarchies. It lets clients treat individual objects and compositions of objects uniformly. In other words, whether dealing with a single object or a group of objects (composite), clients can use them interchangeably.

Components of Composite Design Pattern:

- **Component:** The component declares the interface for objects in the composition and for accessing and managing its child components. This is like a blueprint that tells us what both individual items (leaves) and groups of items (composites) should be able to do. It lists the things they all have in common.
- **Leaf:** Leaf defines behavior for primitive objects in the composition. This is the basic building block of the composition, representing individual objects that don’t have any child components. Leaf elements implement the operations defined by the Component interface.
- **Composite:** Composite stores child components and implements child-related operations in the component interface. This is a class that has child components, which can be either leaf elements or other composites. A composite class implements the methods declared in the Component interface, often by delegating the operations to its child components.
- **Client:** The client manipulates the objects in the composition through the component interface. The client uses the component class interface to interact with objects in the composition structure. If the recipient is a leaf then the request is handled directly. If the recipient is a composite, then it usually forwards the request to its child components, possibly performing additional operations before and after forwarding.

Why do we need Composite Design Pattern?

The Composite Design Pattern was created to address specific challenges related to the representation and manipulation of hierarchical structures in a uniform way. Here are some points that highlight the need for the Composite Design Pattern:

- **Uniform Interface:**
    - The Composite Pattern provides a uniform interface for both individual objects and compositions.
    - This uniformity simplifies client code, making it more intuitive and reducing the need for conditional statements to differentiate between different types of objects.
    - Other design patterns may not offer the same level of consistency in handling individual and composite objects.

- **Hierarchical Structures:**
  - The primary focus of the Composite Pattern is to deal with hierarchical structures where objects can be composed of other objects.
  - While other patterns address different types of problems, the Composite Pattern specifically targets scenarios involving tree-like structures.
- **Flexibility and Scalability:**
  - The Composite Pattern allows for dynamic composition of objects, enabling the creation of complex structures.
  - It promotes flexibility and scalability, making it easier to add or remove elements from the hierarchy without modifying the client code.
- **Common Operations:**
  - By defining common operations at the component level, the Composite Pattern reduces code duplication and promotes a consistent approach to handling both leaf and composite objects.
  - Other design patterns may not provide the same level of support for common operations within hierarchical structures.
- **Client Simplification:**
  - The Composite Pattern simplifies client code by providing a unified way to interact with individual and composite objects. This simplification is particularly valuable when working with complex structures, such as graphical user interfaces or organizational hierarchies.

When to use Composite Design Pattern?

Composite Pattern should be used when clients need to ignore the difference between compositions of objects and individual objects. If programmers find that they are using multiple objects in the same way, and often have nearly identical code to handle each of them, then composite is a good choice, it is less complex in this situation to treat primitives and composites as homogeneous.

- Less number of objects reduces the memory usage, and it manages to keep us away from errors related to memory like java.lang.OutOfMemoryError.
- Although creating an object in Java is really fast, we can still reduce the execution time of our program by sharing objects.

When not to use Composite Design Pattern?

Composite Design Pattern makes it harder to restrict the type of components of a composite. So it should not be used when you don't want to represent a full or partial hierarchy of objects.

- Composite Design Pattern can make the design overly general.
- It makes harder to restrict the components of a composite.
- Sometimes you want a composite to have only certain components. With Composite, you can't rely on the type system to enforce those constraints for you.
- Instead you'll have to use run-time checks.

FACADE DESIGN PATTERN:

The Facade Design Pattern is a structural design pattern that provides a simplified interface to a complex system, making it easier to use. It hides the complexities of the underlying system and provides a single entry point for interacting with it. This pattern is particularly useful when you need to provide a unified interface to a set of interfaces in a subsystem.

- Structuring a system into subsystems helps reduce complexity.
- A common design goal is to minimize the communication and dependencies between subsystems.
- One way to achieve this goal is to introduce a Facade object that provides a single simplified interface to the more general facilities of a subsystem.

Component of Facade Design Pattern:

Facade (Compiler):

- Facade knows which subsystem classes are responsible for a request.
- It delegates client requests to appropriate subsystem objects.

Subsystem classes (Scanner, Parser, ProgramNode, etc.):

- It implements subsystem functionality.
- It handles work assigned by the Facade object.
- It has no knowledge of the facade; that is, they keep no references to it.
- Facade Method Design Pattern collaborate in different way
- Clients communicate with the subsystem by sending requests to Facade, which forwards them to the appropriate subsystem objects.
- The Facade may have to do work of its own to translate its inheritance to subsystem interface.
- Clients that use the Facade don't have to access its subsystem objects directly.

When to use Facade Design Pattern?

A Facade provides a simple default view of the subsystem that is good enough for most clients. Only clients needing more customizability will need to look beyond the facade.

There are many dependencies between clients and the implementation classes of an abstraction.

A Facade to decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability.

Facade defines an entry point to each subsystem level. If subsystems are dependent, then you can simplify the dependencies between them by making them communicate with each other solely through their facades.

Use Cases of Facade Method Design Pattern:

- **Simplifying Complex External Systems:**
    - A facade encapsulates database connection, query execution, and result processing, offering a clean interface to the application.
    - A facade simplifies the usage of external APIs by hiding the complexities of authentication, request formatting, and response parsing.
    - A facade can create a more user-friendly interface for complex or poorly documented libraries.
- **Layering Subsystems:**
    - Decoupling subsystems: Facades define clear boundaries between subsystems, reducing dependencies and promoting modularity.
    - Providing high-level views: Facades offer simplified interfaces to lower-level subsystems, making them easier to understand and use.
- **Providing a Unified Interface to Diverse Systems:**
    - Integrating multiple APIs: A facade can combine multiple APIs into a single interface, streamlining interactions and reducing code duplication.
    - Bridging legacy systems: A facade can create a modern interface for older, less accessible systems, facilitating their integration with newer components.
- **Protecting Clients from Unstable Systems:**
    - Isolating clients from changes: Facades minimize the impact of changes to underlying systems by maintaining a stable interface.
    - Managing third-party dependencies: Facades can protect clients from changes or issues in external libraries or services.

Advantages of Facade Method Design Pattern:

- **Simplified Interface:**

    - Provides a clear and concise interface to a complex system, making it easier to understand and use.
    - Hides the internal details and intricacies of the system, reducing cognitive load for clients.
    - Promotes better code readability and maintainability.
- **Reduced Coupling:**
    - Decouples clients from the underlying system, making them less dependent on its internal structure.
    - Promotes modularity and reusability of code components.
    - Facilitates independent development and testing of different parts of the system.
- **Encapsulation:**
    - Encapsulates the complex interactions within a subsystem, protecting clients from changes in its implementation.
    - Allows for changes to the subsystem without affecting clients, as long as the facade interface remains stable.
- **Improved Maintainability:**
    - Easier to change or extend the underlying system without affecting clients, as long as the facade interface remains consistent.
    - Allows for refactoring and optimization of the subsystem without impacting client code.

Disadvantages of Facade Method Design Pattern:

- **Increased Complexity:**
    - Introducing a facade layer adds an extra abstraction level, potentially increasing the overall complexity of the system.
    - This can make the code harder to understand and debug, especially for developers unfamiliar with the pattern.
- **Reduced Flexibility:**
    - The facade acts as a single point of access to the underlying system.
    - This can limit the flexibility for clients who need to bypass the facade or access specific functionalities hidden within the subsystem.
- **Overengineering:**
    - Applying the facade pattern to very simple systems can be overkill, adding unnecessary complexity where it's not needed.
    - Consider the cost-benefit trade-off before implementing a facade for every situation.
- **Potential Performance Overhead:**
    - Adding an extra layer of indirection through the facade can introduce a slight performance overhead, especially for frequently used operations.
    - This may not be significant for most applications, but it's worth considering in performance-critical scenarios.

FLYWEIGHT DESIGN PATTERN:

The Flyweight Design Pattern is a structural design pattern that aims to minimize memory usage and improve performance by sharing common state between multiple objects. It is particularly useful when you need to create a large number of similar objects that share common characteristics, thus reducing the memory footprint and improving efficiency.One important feature of flyweight objects is that they are immutable. This means that they cannot be modified once they have been constructed.

Why do we care about the number of objects in our program?

- Less number of objects reduces the memory usage, and it manages to keep us away from errors related to memory like java.lang.OutOfMemoryError.

- Although creating an object in Java is really fast, we can still reduce the execution time of our program by sharing objects.

Component of Facade Design Pattern:

- **Flyweight Interface:** Declares the interface for flyweight objects. It includes methods through which flyweight objects can receive and act on extrinsic state.
- **Concrete Flyweight:** Implements the flyweight interface and represents concrete flyweight objects. These objects store intrinsic state (shared among multiple objects) and rely on extrinsic state (passed by the client) to perform operations.
- **Flyweight Factory:** Manages the creation and sharing of flyweight objects. It maintains a pool of existing flyweight objects and provides methods for clients to retrieve or create flyweight objects.
- **Client:** Maintains references to flyweight objects and provides extrinsic state when necessary. Clients use flyweight objects to perform operations, passing extrinsic state as needed.

Use Case of Flyweight Design Patterns:

- **Graphics and Image Processing:** In graphical applications, such as computer games or image processing software, the Flyweight pattern can be used for managing graphical elements like textures, sprites, or fonts. Similar graphics can share a common Flyweight object to save memory.
- **Database Connection Pooling:** When dealing with database connections in applications, the Flyweight pattern can be applied to manage a pool of database connection objects. Instead of creating a new database connection every time one is needed, a shared connection from the pool can be reused, reducing the overhead of creating and closing connections frequently.
- **GUI Systems:** Graphical User Interface (GUI) frameworks can benefit from the Flyweight pattern, especially when dealing with UI elements like buttons, icons, or tooltips. Commonly used UI elements can be shared among different parts of the application, reducing memory consumption.
- **Text Processing Applications:** In applications that involve processing and analyzing large amounts of text, the Flyweight pattern can be applied to represent shared elements such as words, phrases, or linguistic constructs. This can lead to more efficient use of memory and improved performance.
- By applying the Flyweight pattern in these scenarios, developers can achieve a balance between memory efficiency and performance, especially in situations where a large number of similar objects need to be managed.

Advantages of Flyweight Design Patterns:

- **Memory Efficiency:** One of the primary advantages of the Flyweight pattern is its ability to reduce memory usage. It achieves this by sharing common parts of objects, rather than duplicating them across multiple instances.
- **Performance Improvement:** By sharing common state across multiple objects, the Flyweight pattern can lead to performance improvements. This is particularly beneficial in situations where creating and managing a large number of objects would be computationally expensive.
- **Resource Conservation:** The Flyweight pattern helps conserve resources by avoiding the redundant storage of shared data. Instead of each instance holding its own copy of shared data, they reference a shared instance of that data.
- **Object-Oriented Principle Compliance:** The Flyweight pattern promotes the separation of intrinsic and extrinsic state. Intrinsic state is shared and stored in the flyweight objects,

while extrinsic state is passed in when needed. This aligns with the object-oriented design principles of encapsulation and separation of concerns.

- **Improved Maintainability:** Since the Flyweight pattern promotes the reuse of shared objects, changes to the shared state can be centralized and applied to all instances. This can make maintenance easier, as modifications are concentrated in a smaller set of objects.
- **Enhanced Scalability:** The Flyweight pattern is particularly useful in scenarios where a large number of objects need to be created dynamically. It allows systems to scale more effectively by reducing the memory and processing overhead associated with creating and managing numerous objects.
- **Applicability in GUI and Graphics:** The Flyweight pattern is often employed in graphical applications where numerous similar graphical elements need to be displayed. For example, in a drawing application, individual characters in a text document might be implemented as flyweight objects.

Disadvantages of Flyweight Design Patterns:

- **Increased Complexity:** Introducing the Flyweight pattern may add complexity to the code, especially in scenarios where the separation of intrinsic and extrinsic state is not straightforward. This complexity can make the code harder to understand and maintain.
- **Potential for Overhead:** In some cases, the overhead associated with managing and looking up shared flyweight objects may outweigh the benefits gained from memory savings. This is especially true when the number of instances is relatively small or when the cost of creating and managing the flyweight objects is high.
- **Limited Applicability:** The Flyweight pattern is most effective when there is a significant amount of shared state among objects. In situations where objects are mostly unique and have little shared state, the benefits of the Flyweight pattern may be limited.
- **Thread Safety Concerns:** If multiple threads are involved and modifications to shared flyweight objects are possible, additional synchronization mechanisms may be needed to ensure thread safety. This can introduce complexity and potentially impact performance.
- **Difficulty in Managing Extrinsic State:** Managing the extrinsic state (state that is not shared) of flyweight objects can be challenging, especially if it involves complex interactions. Developers need to carefully manage the extrinsic state to avoid introducing errors or unexpected behavior.
- **Impact on Identity:** The Flyweight pattern blurs the identity of objects by sharing their internal state. This can be problematic if the identity of individual objects is important in the application logic. For example, if two flyweight objects are considered equal based on their shared state, it might lead to unexpected behavior.
- **Design Rigidity:** Introducing the Flyweight pattern may make the code more rigid, especially if the shared state needs to be modified or extended. Changes to the shared state may require modifications to multiple objects, potentially impacting the flexibility of the design.

PROXY DESIGN PATTERN:

The Proxy Design Pattern is a structural design pattern that provides a surrogate or placeholder for another object to control access to it. It allows you to create a proxy object that acts as an intermediary between the client and the real object, enabling you to add functionality such as lazy initialization, access control, logging, or caching without changing the existing code of the real object.

Components of Proxy Design Pattern:

- **Subject:**
    - The Subject is an interface or an abstract class that defines the common interface shared by the RealSubject and Proxy classes.

    - It declares the methods that the Proxy uses to control access to the RealSubject.
    - Declares the common interface for both RealSubject and Proxy.
- Usually includes the methods that the client code can invoke on the RealSubject and the Proxy.
- **RealSubject:**
    - The RealSubject is the actual object that the Proxy represents. It contains the real implementation of the business logic or the resource that the client code wants to access.
    - It Implements the operations declared by the Subject interface.
- Represents the real resource or object that the Proxy controls access to.
- **Proxy:**
    - The Proxy acts as a surrogate or placeholder for the RealSubject. It controls access to the real object and may provide additional functionality such as lazy loading, access control, or logging.
    - Implements the same interface as the RealSubject (Subject).
- Maintains a reference to the RealSubject.
    - Controls access to the RealSubject, adding additional logic if necessary.
- **Client:**
    - The client code interacts with the Proxy as if it were the real object. The client is unaware of whether it's using the real object or a proxy.

Why do we need Proxy Design Pattern?

The Proxy Design Pattern is employed to address various concerns and scenarios in software development, providing a way to control access to objects, add functionality, or optimize performance.

- **Lazy Loading:**
    - One of the primary use cases for proxies is lazy loading. In situations where creating or initializing an object is resource-intensive, the proxy delays the creation of the real object until it is actually needed.
    - This can lead to improved performance by avoiding unnecessary resource allocation.
- **Access Control:**
    - Proxies can enforce access control policies.
    - By acting as a gatekeeper to the real object, proxies can restrict access based on certain conditions, providing security or permission checks.
- **Protection Proxy:**
    - Protection proxies control access to a real object by adding an additional layer of security checks.
    - They can ensure that the client code has the necessary permissions before allowing access to the real object.
- **Caching:**
    - Proxies can implement caching mechanisms to store results or resources.
    - This is particularly useful when repeated operations on a real object can be optimized by caching previous results, avoiding redundant computations or data fetching.
- **Logging and Monitoring:**
    - Proxies provide a convenient point to add logging or monitoring functionalities.
    - By intercepting method calls to the real object, proxies can log information, track usage, or measure performance without modifying the real object.

When to use Proxy Design Pattern?

- **Deferred Object Creation:**

    - Use a proxy when you want to postpone the creation of a resource-intensive object until it's actually needed.
    - This helps in optimizing the application's startup time and resource usage.
- **Access Control and Permissions:**
    - Use a proxy when you need to control and manage access to an object, ensuring that certain conditions or permissions are met before allowing clients to interact with the real object.
    - This is particularly useful for enforcing security measures.
- **Resource Optimization:**
    - Use a proxy to optimize the utilization of resources, such as caching results or storing previously fetched data.
    - This can lead to performance improvements by avoiding redundant computations or data retrieval.
- **Remote Object Interaction:**
    - Use a proxy when dealing with distributed systems and you want to interact with objects located in different addresses or systems.
    - The proxy can handle the communication details, making remote object interaction more seamless.

When not to use Proxy Design Pattern?

- **Overhead for Simple Operations:** Avoid using a proxy for simple objects or operations that don't involve resource-intensive tasks. Introducing a proxy might add unnecessary complexity in such cases.
- **Unnecessary Abstraction:** If your application doesn't require lazy loading, access control, or additional functionalities provided by proxies, introducing proxies may lead to unnecessary abstraction and code complexity.
- **Performance Impact:** If the introduction of a proxy negatively impacts performance rather than improving it, especially in cases where objects are lightweight and creation is not a significant overhead.
- **When Access Control Isn't Needed:** If there are no access control requirements and the client code can directly interact with the real object without any restrictions.
- **When Eager Loading is Acceptable:** If eager loading of objects is acceptable and doesn't affect the performance of the system, introducing a proxy for lazy loading might be unnecessary.

Use Cases of Proxy Method Design Pattern:

- **Lazy Loading:** You can use a proxy to implement lazy loading, where the real object is created and initialized only when it is accessed for the first time. This is beneficial when creating the real object is resource-intensive or time-consuming.
- **Logging:** Proxy can be employed for logging method calls, providing a way to log information such as the method name, parameters, and results. This can be helpful for debugging or auditing purposes.
- **Caching:** You can use a proxy to implement caching mechanisms. The proxy can check whether the result of a method call is already cached and return the cached result instead of invoking the real object.
- **Virtual Proxy:** A virtual proxy can be used to represent expensive objects, such as large images or complex calculations, without loading them into memory until they are actually needed.
- **Protection Proxy:** This type of proxy controls access to sensitive operations. For instance, you might use a protection proxy to check if the client has the necessary permissions before allowing a specific operation.

- **Remote Proxy:** A remote proxy can be employed when dealing with distributed systems. It acts as a local representative for an object that resides in a different address space, providing a way to interact with it remotely.
- **Monitoring and Metrics:** Gather metrics or monitoring data by intercepting method calls. This can be used to collect information about the usage patterns of specific functionalities.

Advantages of the Proxy Method Design Pattern:

- **Improved Performance:** Proxies improves the performance by controlling the loading and initialization of objects, especially in scenarios where certain operations are resource-intensive.
- **Separation of Concerns:** The Proxy Design Pattern promotes a separation of concerns by isolating the proxy-specific functionality from the real object's implementation. This enhances code modularity and maintainability.
- **Easy Testing:** Proxies can be used to create mock objects for testing purposes. This facilitates unit testing by isolating the unit under test from its dependencies, making it easier to control and verify behavior.
- **Dynamic Behavior:** Proxies can dynamically alter the behavior of the real object without changing its code. This allows additional functionalities, such as logging or monitoring, without modifying the original implementation.
- **Reduced Resource Usage:** Virtual proxies restrict the creation of expensive objects until they are needed. This reduces resource consumption and optimizes memory usage.
- **Enhanced Security:** Proxy objects can add an additional layer of security by implementing authentication, authorization, or other security-related checks before allowing access to the real object.

Disadvantages of the Proxy Method Design Pattern:

- **Complexity:** Introducing proxies can make the codebase complex, especially if multiple types of proxies are involved.
- **Overhead:** Proxies may result in extra overhead, especially if there is significant logic in the proxy class, such as logging, monitoring, or security checks. This overhead might impact the overall performance of the system.
- **Increased Development Time:** Introducing proxies might require additional development time and effort. Developers need to carefully design and implement the proxy logic, and this can be time-consuming.
- **Misuse or Overuse:** There is a risk of misusing or overusing the Proxy Design Pattern. If proxies are applied unnecessarily, it might lead to unnecessary complexity and decreased code readability.
- **Potential Security Risks:** Security-related proxies might introduce new risks if not implemented correctly. For example, a security proxy might unintentionally allow unauthorized access.
- **Maintenance Challenges:** If not designed and documented properly, the maintenance of proxies and their interactions with real objects can become difficult. Changes to one part of the system may require updating multiple proxy classes.
- **Difficulty in Debugging:** The presence of proxy layers can make debugging more challenging, because to find and fix problems, developers must navigate both proxy and real object behavior.

NULL OBJECT DESIGN PATTERN:

The Null Object Design Pattern is a behavioral design pattern that provides an object as a substitute for null references. Instead of using a null reference to represent the absence of an object, the pattern uses a special null object that implements the expected interface but does nothing or provides default behavior. This is achieved by using instances of a concrete class that implements a known interface, instead of null references. We create an abstract class specifying various operations to be done, concrete classes extending this class and a null object class providing do nothing implementation of this class and will be used seamlessly where we need to check null value.

Components of Null Object Design Pattern:

- **Client:** This class has a dependency that may or may not be required. Where no functionality is required in the dependency, it will execute the methods of a null object.
- **DependencyBase:** This abstract class is the base class for the various available dependencies that the Client may use. This is also the base class for the null object class. Where the base class provides no shared functionality, it may be replaced with an interface.
- **Dependency:** This class is a functional dependency that may be used by the Client.
- **NullObject:** This is the null object class that can be used as a dependency by the Client. It contains no functionality but implements all of the members defined by the DependencyBase abstract class.

Advantages:

- It defines class hierarchies consisting of real objects and null objects. Null objects can be used in place of real objects when the object is expected to do nothing. Whenever client code expects a real object, it can also take a null object.
- Also makes the client code simple. Clients can treat real collaborators and null collaborators uniformly. Clients normally don't know whether they're dealing with a real or a null collaborator. This simplifies client code, because it avoids having to write testing code which handles the null collaborator specially.

Disadvantages:

- Can be difficult to implement if various clients do not agree on how the null object should do nothing as when your AbstractObject interface is not well defined.
- Can necessitate creating a new NullObject class for every new AbstractObject class.

INTERPRETER DESIGN PATTERN:

The Interpreter Design Pattern is a behavioral design pattern that defines a grammar for a language and provides a way to interpret and evaluate sentences in that language, such as SQL queries, mathematical expressions, or regular expressions.

- It provides a mechanism to evaluate sentences in a language by representing their grammar as a set of classes. Each class represents a rule or expression in the grammar, and the pattern allows these classes to be composed hierarchically to interpret complex expressions.
- The pattern involves defining a hierarchy of expression classes, both terminal and nonterminal, to represent the elements of the language's grammar.
- Terminal expressions represent basic building blocks, while nonterminal expressions represent compositions of these building blocks.
- The tree structure of the Interpreter design pattern is somewhat similar to that defined by the composite design pattern with terminal expressions being leaf objects and non-terminal expressions being composites.

Components of the Interpreter Design Pattern:

- **AbstractExpression:**
  - This is an abstract class or interface that declares an abstract interpret() method. It represents the common interface for all concrete expressions in the language.
- **TerminalExpression:**
  - These are the concrete classes that implement the AbstractExpression interface. Terminal expressions represent the terminal symbols or leaves in the grammar.

These are the basic building blocks that the interpreter uses to interpret the language.
    - For example, in an arithmetic expression interpreter, terminal expressions could include literals such as numbers or variables representing numeric values.
    - These terminal expressions would evaluate to their respective values directly without further decomposition.
- **NonterminalExpression:**
    - These are also concrete classes that implement the AbstractExpression interface. Non-terminal expression classes are responsible for handling composite expressions, which consist of multiple sub-expressions. These classes are tasked to provide the interpretation logic for such composite expressions.
    - Another aspect of non-terminal expressions is their responsibility to coordinate the interpretation process by coordinating the interpretation of sub-expressions.
    - This involves coordinating the interpretation calls on sub-expressions, aggregating their results, and applying any necessary modifications or operations to achieve the final interpretation of the entire expression
    - Non-terminal expressions facilitate the traversal of expression trees during the interpretation process.
    - As part of this traversal, they recursively interpret their sub-expressions, ensuring that each part of the expression contributes to the overall interpretation.
- **Context:**
    - This class contains information that is global to the interpreter and is maintained and modified during the interpretation process. The context may include variables, data structures, or other state information that the interpreter needs to access or modify while interpreting expressions.
- **Client:**
    - The client is responsible for creating the abstract syntax tree (AST) and invoking the interpret() method on the root of the tree. The AST is typically created by parsing the input language and constructing a hierarchical representation of the expressions.
- **Interpreter:**
    - The interpreter is responsible for coordinating the interpretation process. It manages the context, creates expression objects representing the input expression, and interprets the expression by traversing and evaluating the expression tree. The interpreter typically encapsulates the logic for parsing, building the expression tree, and interpreting the expressions according to the defined grammar.

Benefits of using the Interpreter Pattern:

- **Modularity:** Components such as terminal and non-terminal expressions can be easily added or modified to support new language constructs or operations.
- **Separation of Concerns:** The pattern separates the grammar interpretation from the client, allowing the client to focus on providing input expressions while leaving the interpretation logic to the interpreter components.
- **Extensibility:** New operations or language constructs can be added without modifying existing code, promoting code reuse and maintainability.

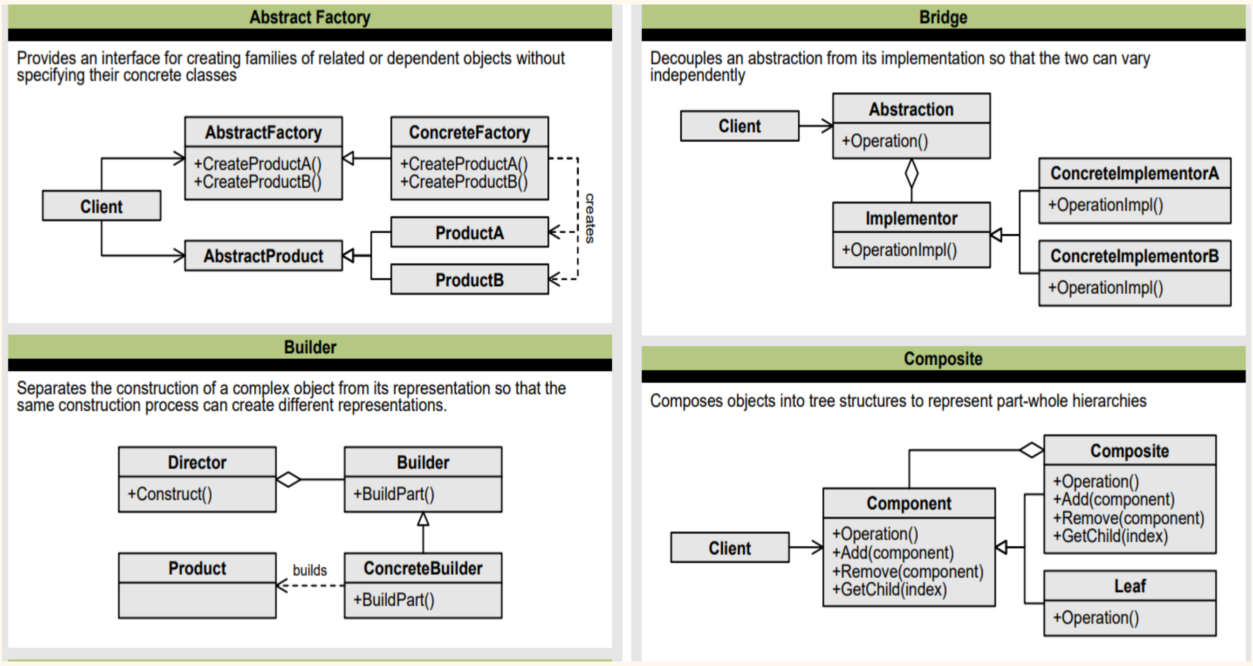
When to use Interpreter Design Pattern:

- **When dealing with domain-specific languages:** If you need to interpret and execute expressions or commands in a domain-specific language (DSL), the Interpreter pattern can provide a flexible and extensible way to implement the language's grammar and semantics.

- **When you have a grammar to interpret:** If you have a well-defined grammar for expressions or commands that need to be interpreted, the Interpreter pattern can help parse and evaluate these structures efficiently.
- **When adding new operations is frequent:** If your application frequently requires the addition of new operations or commands, the Interpreter pattern allows you to add new expression classes easily without modifying existing code, thus promoting maintainability and extensibility.
- **When you want to avoid complex grammar parsers:** If building and maintaining complex grammar parsers seems daunting or unnecessary for your use case, the Interpreter pattern offers a simpler alternative for interpreting expressions directly.

When not to use Interpreter Design Pattern?

- **For simple computations:** If your task involves only simple computations or operations that can be easily handled by built-in language features or libraries, using the Interpreter pattern may introduce unnecessary complexity.
- **When performance is critical:** Interpreting expressions through the Interpreter pattern might introduce overhead compared to other approaches, especially for complex expressions or large input sets. In performance-critical applications, a more optimized solution, such as compilation to native code, may be preferable.
- **When the grammar is too complex:** If your grammar is highly complex, with numerous rules and exceptions, implementing it using the Interpreter pattern may lead to a proliferation of expression classes and increased code complexity. In such cases, a dedicated parser generator or compiler may be more suitable.
- **When there’s no need for extensibility:** If the requirements of your application are fixed and well-defined, and there’s no anticipation of adding new operations, commands, or language constructs in the future, then implementing the Interpreter pattern may introduce unnecessary complexity.



Abstract Factory

Provides an interface for creating families of related or dependent objects without specifying their concrete classes

Bridge

Decouples an abstraction from its implementation so that the two can vary independently

Builder

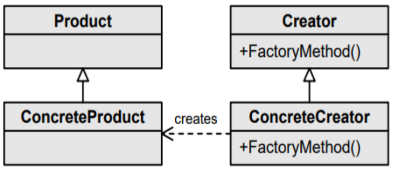
Separates the construction of a complex object from its representation so that the same construction process can create different representations.

Composite

Composes objects into tree structures to represent part-whole hierarchies

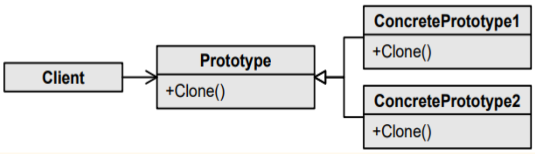
Factory Method

Defines an interface for creating an object but let subclasses decide which class to instantiate



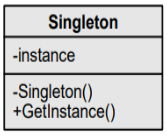
Prototype

Specifies the kinds of objects to create using a prototypical instance and create new objects by copying this prototype



Singleton

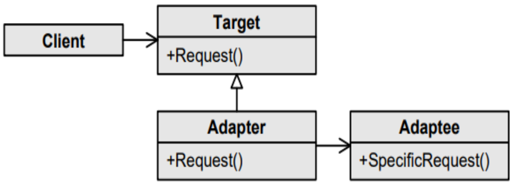
Ensure a class only has one instance and provide a global point of access to it



Structural Patterns

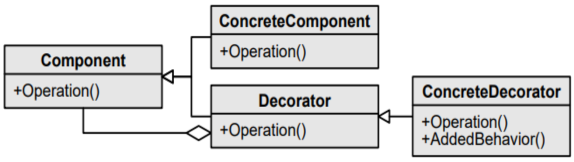
Adapter

Converts the interface of a class into another interface clients expect



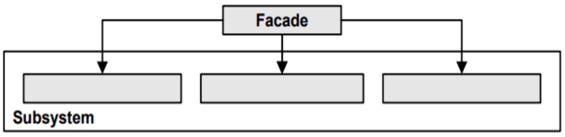
Decorator

Attaches additional responsibilities to an object dynamically



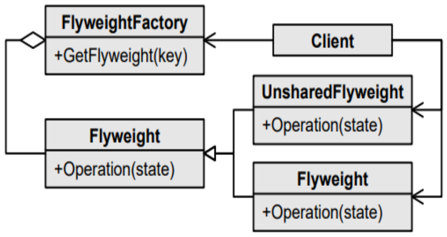
Facade

Provides a unified interface to a set of interfaces in a subsystem



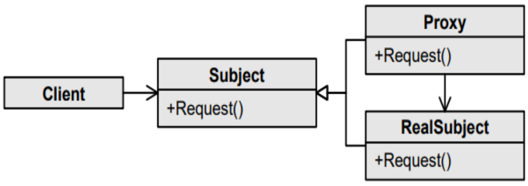
Flyweight

Uses sharing to support large numbers of fine-grained objects efficiently



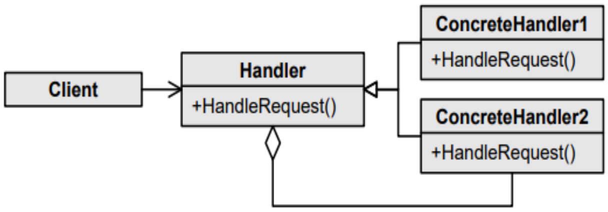
Proxy

Provides a surrogate or placeholder for another object to control access to it



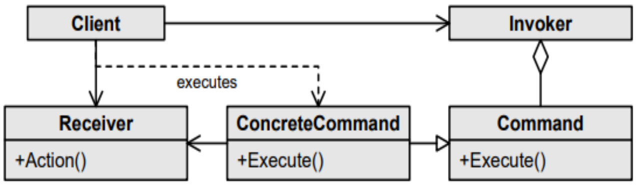
Chain of Responsibility

Avoids coupling the sender of a request to its receiver by giving more than one object a chance to handle the request



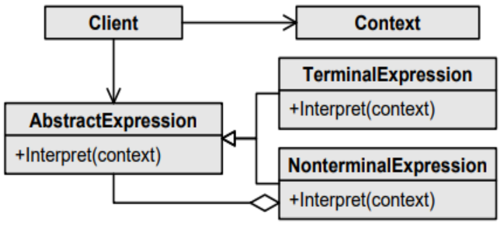
Command

Encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations



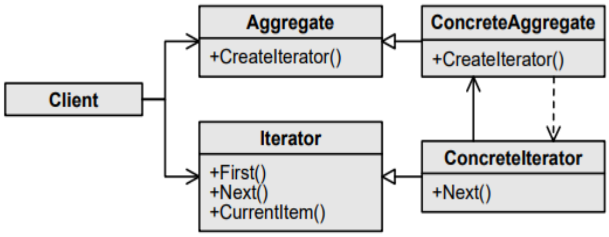
Interpreter

Given a language, defines a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language



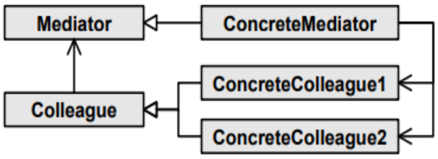
Iterator

Given a language, defines a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language



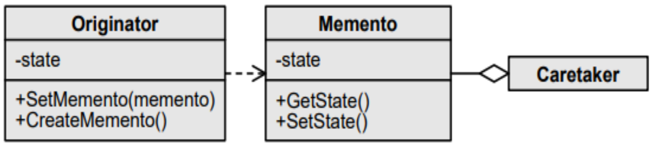
Mediator

Defines an object that encapsulates how a set of objects interact



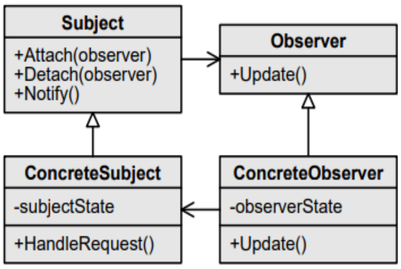
Memento

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later



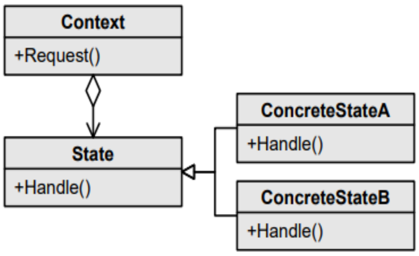
Observer

Defines a one-to-many dependency between objects so that when one object changes state all its dependents are notified and updated automatically



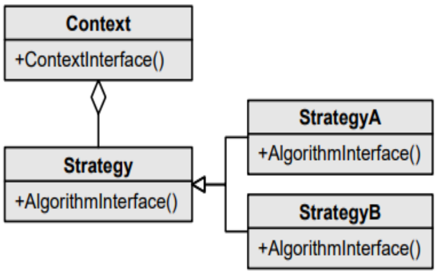
State

Allows an object to alter its behavior when its internal state changes



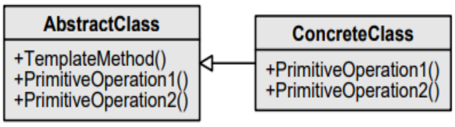
Strategy

Defines a family of algorithms, encapsulate each one, and make them interchangeable



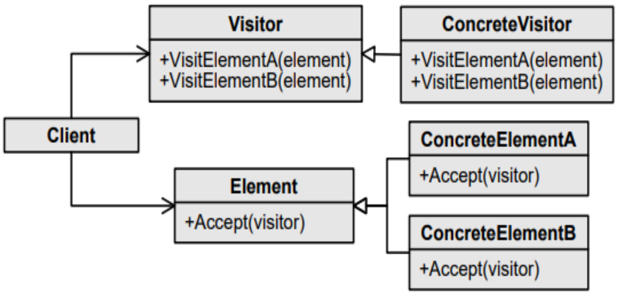
TemplateMethod

Defines the skeleton of an algorithm in an operation, deferring some steps to subclasses



TemplateMethod

Represents an operation to be performed on the elements of an object structure


Visitor
+VisitElementA(element)
+VisitElementB(element)
ConcreteVisitor
+VisitElementA(element)
+VisitElementB(element)
Client
ConcreteElementA
+Accept(visitor)
Element
+Accept(visitor)
ConcreteElementB
+Accept(visitor)