1. What is Hashing?
2. What is Consistent Hashing?
3. What is Bloom Filters?

**Answer.**

***Problem:***

If we have a large set of structured data (identified by record IDs) stored in a set of data files, what is the most efficient way to know which file might contain our required data? We don’t want to read each file, as that would be slow, and we have to read a lot of data from the disk. One solution can be to build an index on each data file and store it in a separate index file. This index can map each record ID to its offset in the data file. Each index file will be sorted on the record ID. Now, if we want to search an ID in this index, the best we can do is a Binary Search. Can we do better than that?

***Solution:***

1. What is the concept of Leader and Follower?

**Answer.**

***Problem:***

Distributed systems keep multiple copies of data for fault tolerance and higher availability. A system can use **quorum** to ensure data consistency between replicas, i.e., all reads and writes are not considered successful until a majority of nodes participate in the operation. However, using quorum can lead to another problem, that is, lower availability; at any time, the system needs to ensure that at least a majority of replicas are up and available, otherwise the operation will fail. Quorum is also not sufficient, as in certain failure scenarios, the client can still see inconsistent data.

***Solution:***

Allow only a single server (called leader) to be responsible for data replication and to coordinate work.

At any time, one server is elected as the leader. This leader becomes responsible for data replication and can act as the central point for all coordination. The followers only accept writes from the leader and serve as a backup. In case the leader fails, one of the followers can become the leader. In some cases, the follower can serve read requests for load balancing.

***Real world applications:***

In **Kafka**, each partition has a designated leader which is responsible for all reads and writes for that partition. Each follower’s responsibility is to replicate the leader’s data to serve as a “backup” partition. This provides redundancy of messages in a partition, so that a follower can take over the leadership if the leader goes down.

To ensure strong consistency, **Paxos** (hence Chubby) performs leader election at start up. This leader is responsible for data replication and coordination.

1. What is multi leader replication?
2. What is Leaderless replication?
3. What is write Ahead Log (WAL)?

**Answer.**

***Problem:***

Machines can fail or restart anytime. If a program is in the middle of performing a data modification, what will happen when the machine it is running on loses power? When the machine restarts, the program might need to know the last thing it was doing. Based on its atomicity and durability needs, the program might need to decide to redo or undo or finish what it had started. How can the program know what it was doing before the system crash?

1. What is Segmented Log?
2. B tree and B+ tree
3. B tree vs LSM
4. What in inverted index?

**Answer.**

An inverted index is called ***"inverted"*** because it inverts the relationship between the words and the documents, compared to a traditional index. **In a traditional index**, each document is associated with a set of words. In an **inverted index**, each word is associated with a set of documents.

**For example**, suppose you have a collection of documents that contain the following words:

Document 1: "dog cat bird"

Document 2: "bird dog"

Document 3: "cat mouse"

A traditional index of these documents would look something like this:

Document 1: "dog", "cat", "bird"

Document 2: "bird", "dog"

Document 3: "cat", "mouse"

In contrast, an inverted index would look like this:

"bird": [1, 2]

"cat": [1, 3]

"dog": [1, 2]

"mouse": [3]

The inverted index maps each word to a list of document IDs in which the word appears. This allows you to quickly determine which documents contain a particular word, making it faster to search for documents that match a query.

In general, inverted indexes are faster for searching for terms within a large collection of documents, as they allow for efficient searching and retrieval of documents that match a particular set of keywords.

***Real World Application Example:***

Elasticsearch heavily relies on inverted indices to index text fields. It makes the searching extremely fast. When a document is stored, it is indexed and fully searchable in [near real-time](https://www.elastic.co/guide/en/elasticsearch/reference/current/near-real-time.html)--**within 1 second.**

1. What is Gossip protocol?

**Answer.**

***Problem:***

In distributed systems, maintaining consistent state across all nodes is a challenging problem. One common approach is to use a centralized node to coordinate all updates to the system. However, this can lead to a single point of failure, reduced availability, and limited scalability.

What if we do not have any central node that keeps track of all nodes to know if a node is down or not, how does a node know every other node’s current state? The simplest way to do this is to have every node maintain a heartbeat with every other node. Then, when a node goes down, it will stop sending out heartbeats, and everyone else will find out immediately. But, this means messages get sent every tick (being the total number of nodes), which is a ridiculously high amount and will consume a lot of network bandwidth.

***Solution :***

One way to achieve the consistency across all nodes is by using a gossip protocol. Gossip protocol is Peer to Peer(P2P) communication mechanism. Gossip is a way for nodes to exchange information with each other by randomly selecting other nodes to share information with. Each node periodically sends its own state to a small, random set of other nodes, which then update their own state accordingly. Over time, information about the state of the system spreads throughout the network, and all nodes eventually converge to a consistent state.

Gossip protocols are useful in situations where a central coordinator is not desirable or practical, and where the system state is too large to be efficiently propagated by broadcasting to all nodes. They are commonly used in distributed databases, where they help ensure that all nodes have the same view of the database, and in peer-to-peer file sharing systems, where they help ensure that all nodes have access to the same files.

***Real World Application Example:***

**Dynamo & Cassandra** use gossip protocol which allows each node to keep track of state information about the other nodes in the cluster, like which nodes are reachable, what key ranges they are responsible for, etc.

1. What is CAP theorem?
2. What is PARCEL theorem?
3. Different cache mechanism ?
4. What is Merkel tree?

**Answer.**

***Problem:***

Read Repair removes conflicts while serving read requests. But, if a replica falls significantly behind others, it might take a very long time to resolve conflicts. It would be nice to be able to automatically resolve some conflicts in the background. To do this, we need to quickly compare two copies of a range and figure out exactly which parts are different. In a distributed environment, how can we quickly compare two copies of a range of data residing on two different replicas and figure out exactly which parts are different?

A replica can contain a lot of data. Naively splitting up the entire range to calculate checksums for comparison, is not very feasible; there is simply too much data to be transferred. Instead, we can use Merkle trees to compare replicas of a range.

A Merkle tree is a binary tree of hashes, where each internal node is the hash of its two children, and each leaf node is a hash of a portion of the original data.

***How the data is compared using Merkel Tree?***

1. Compare the root hashes of both trees.
2. If they are equal, stop.
3. Recurse on the left and right children.

This means that replicas know exactly which parts of the range are different, but the amount of data exchanged is minimized. The principal advantage of a Merkle tree is that each branch of the tree can be checked independently without requiring nodes to download the entire tree or the entire data set. Hence, Merkle trees minimize the amount of data that needs to be transferred for synchronization and reduce the number of disk reads.

***Real World Application Example:***

Apache Cassandra uses Merkel trees to implement its distributed write protocol, which ensures that data written to multiple nodes is consistent even in the presence of network partitions and node failures.

1. What is Quorum?

**Answer.**

***Problem :***

In Distributed Systems, data is replicated across multiple servers for fault tolerance and high availability. Once a system decides to maintain multiple copies of data, another problem arises: how to make sure that all replicas are consistent, i.e., if they all have the latest copy of the data and that all clients see the same view of the data?

***Solution :***

Quorum is a term used in distributed systems to describe the minimum number of nodes or replicas in a system that must agree on a particular operation for it to be considered successful. In a quorum-based system, a specific number of nodes must respond positively to a request in order for the operation to be considered valid.

***Example:***

Consider a distributed database with three replicas, A, B, and C. If a write operation is performed on replica A and a **quorum of 2 is required**, then replicas B and C must also confirm the write before it can be considered successful. In this case, the write operation is considered committed only when at least 2 out of the 3 replicas agree on it.

Similarly, in a read operation, a quorum can be used to ensure that the data being read is consistent across the replicas. For example, if a quorum of 2 is required for a read operation, the system will only return data if at least 2 out of the 3 replicas have the same data.

***What value should be used for Quorum?***

More than half of the number of nodes in the cluster: (*N/2 + 1*) where N is the total number of nodes in the cluster.

***For Example:***

* In a ***5-node***cluster, 3 nodes must be online to have a majority.
* In a ***4-node***cluster, 3 nodes must be online to have a majority.
* With ***5*-node**, the system can afford ***2 node*** failures, whereas,

with ***4-node***, it can afford only ***1 node***failure.

Because of this logic, it is recommended to always have an odd number of total nodes in the cluster. Quorum is achieved when nodes follow the below protocol: *R + W > N*, *Where*,

* **N** = nodes in the quorum group
* **W** = minimum write nodes
* **R** = minimum read node

***Real World Application’s Example:***

For leader election, **Chubby** uses Paxos, which use quorum to ensure strong consistency. As stated above, quorum is also used to ensure that at least one node receives the update in case of failures. For instance, in **Cassandra**, to ensure data consistency, each write request can be configured to be successful only if the data has been written to at least a quorum (or majority) of replica nodes.

1. What are the different types of Quorum?

**Answer.**

There are several types of quorums that are used in different distributed systems:

Majority quorum: A majority quorum is the most common type of quorum. It requires a majority of the nodes to agree on a decision in order to reach consensus.

Strict quorum: A strict quorum requires that a certain set of nodes be reachable in order to reach consensus. For example, a strict quorum with n nodes might require that exactly n/2 + 1 nodes be reachable.

Write quorum: A write quorum requires a certain number of nodes to agree on a write operation in order to make the write durable.

Read quorum: A read quorum requires a certain number of nodes to agree on a read operation in order to ensure consistency.

Sloppy quorum: A sloppy quorum allows for a certain number of failures among the nodes, as long as a quorum of nodes can still be reached.

1. What is Sloppy Quorum and why is it used?

***Problem :***

A network problem can cause a client to lose connection with many of the nodes in a database. Even though these nodes are still working and other clients can still connect to them, the disconnected client cannot access them. In this case, it's possible that there are fewer than the required number of reachable nodes (w or r), so the client can't form a quorum. However, in a large cluster with many more than n nodes, it's likely that the client can still connect to some of the database nodes, even during a network interruption, just not the ones needed to reach a quorum for a specific value.

***Solution :***

In a Sloppy quorum, nodes are allowed to operate even if they can't communicate with a certain number of other nodes. This is different from a strict quorum, where nodes must be able to communicate with a majority of other nodes in order to operate.

The main reason to use a sloppy quorum is to provide more flexibility and availability in the face of node failures or network partitions. In a strict quorum, a single node failure or network partition could prevent the system from functioning. But in a sloppy quorum, the system can continue to operate as long as a sufficient number of nodes are still able to communicate with each other.

In Sloppy Quorum, write and reads still require w and r successful responses, but those may include nodes that are not among the designated n “home” nodes for a value.

Once the network interruption is fixed, any writes that one node temporarily accepted on behalf of another node are sent to the appropriate “home” nodes.

A Sloppy Quorum example with **10 nodes** would work as follows:

• The number of "home" nodes for a value is designated as "n". In this case, n = 10.

• The number of successful write responses required is designated as "w". In this example, let's say w = 7.

• The number of successful read responses required is designated as "r". In this example, let's say r = 5.

When a client wants to write a value, it sends the write request to 7 nodes. The write is considered successful if at least 7 of those nodes respond and confirm that the write has been made. In a Sloppy Quorum, the 7 nodes that respond and confirm the write do not have to be among the designated "home" nodes of 10. They can be any 7 nodes in the cluster.

When a client wants to read a value, it sends the read request to 5 nodes. The read is considered successful if at least 5 of those nodes respond and return the value. In a Sloppy Quorum, the 5 nodes that respond do not have to be among the designated "home" nodes of 10. They can be any 5 nodes in the cluster.

***Example :***

Sloppy quorums are optional in all common Dynamo implementations. In Riak, they are enabled by default, and in Cassandra they are disabled by default.

1. What is Hinted Handoff?

**Answer.**

***Problem :***

Depending upon the consistency level, a distributed system can still serve write requests even when nodes are down. For example, if we have the replication factor of three and the client is writing with a quorum consistency level. This means that if one of the nodes is down, the system can still write on the remaining two nodes to fulfil the consistency level, making the write successful. Now, when the node which was down comes online again, how should we write data to it?

***Solution :***

The technique used to solve this problem is called “Hinted Handoff”. The main problem that hinted handoff solves is ensuring that updates to a node's data are not lost when the node becomes temporarily unavailable. Without hinted handoff, updates would be lost if the node became unavailable and there was no other node available to receive the updates. With hinted handoff, updates can be temporarily stored on another node until the original node becomes available again, ensuring that the updates are not lost.

When a node is down or is not responding to a write request, the node which is coordinating the operation, writes a hint in a text file on the local disk. This hint contains the data itself along with information about which node the data belongs to. When the coordinating node discovers that a node for which it holds hints has recovered, it forwards the write requests for each hint to the target.

***Hinted Hanoff with Sloppy Quorum:***

In a Sloppy Quorum, when a node becomes unavailable, other nodes take over its responsibilities and continue serving requests. This is known as hinted handoff. When a write request is made to a node that is unavailable, the coordinator node for the request stores the write request as a "hint" in a temporary cache on a node that is still available. The hint is then sent to the original node once it becomes available again. The hinted node can then apply the write request and update its own data accordingly.

The advantage of using a hinted handoff with a Sloppy Quorum is that it allows the system to continue serving write requests even when some nodes are unavailable. This helps to maintain data consistency and minimize downtime.

It's important to note that hinted handoff is not a replacement for proper node management and maintenance. The hints should be temporary and the original node should be brought back online as soon as possible. Hinted handoff is simply a mechanism for preserving data consistency during short-term outages.

***Example :***

* Cassandra nodes use Hinted Handoff to remember the write operation for failing nodes.
* Dynamo ensures that the system is “always-writeable” by using Hinted Handoff (and Sloppy Quorum)

1. What is Checksum?

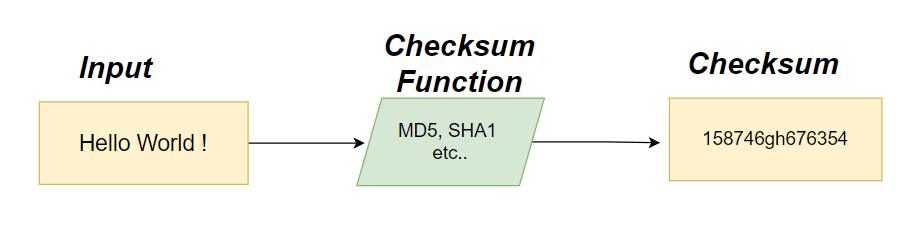
**Answer.**

***Problem :***

In a distributed system, while moving data between components, it is possible that the data fetched from a node may arrive corrupted. This corruption can occur because of faults in a storage device, network, software, etc. How can a distributed system ensure data integrity, so that the client receives an error instead of corrupt data?

***Explanation :***

Calculate a checksum and store it with data. To calculate a checksum, a cryptographic hash function like MD5, SHA-1, SHA-256, or SHA-512 is used. The hash function takes the input data and produces a string (containing letters and numbers) of fixed length; this string is called the checksum.



When a system is storing some data, it computes a checksum of the data, and stores the checksum with the data. When a client retrieves data, it verifies that the data it received from the server matches the checksum stored. If not, then the client can opt to retrieve that data from another replica.

***Example :***

**HDFS and Chubby store** the checksum of each file with the data

1. What is Database isolation level?

**Answer.**

In the context of databases, isolation level refers to the degree to which the changes made by one transaction are isolated from the changes made by other transactions. Isolation levels are used to control the concurrent access to data in a database, and are typically set at the session or transaction level. The most common isolation levels are:

**Read Uncommitted**: This is the **lowest isolation level**, and allows transactions to read data that is currently being modified by other transactions, leading to "dirty reads."

**Read Committed:** This isolation level allows transactions to read data that has been committed by other transactions, but not data that is currently being modified. This level eliminates "dirty reads," but may still lead to "non-repeatable reads" or "phantom reads."

**Repeatable Read:** This isolation level ensures that a transaction will see the same data each time it reads the same data, but it still allows other transactions to insert new rows that match the search criteria of a query.

**Serializable:** This is the **highest isolation level**, and ensures that a transaction can only read or modify data that has been previously committed by other transactions, and that all data changes made by the transaction are completely isolated from other transactions.

1. What’s the concept of Read Repair?

**Answer.**

***Problem:***

In a distributed database system, data is typically replicated across multiple nodes to ensure high availability and fault tolerance. However, this replication can lead to inconsistencies in the data if updates are made to different replicas at different times. Let’s assume a scenario where a node wasn’t able to get the latest write or update request because the node was down. When a client reads data from this replica, it may receive stale data that is inconsistent with the data stored on other replicas Now, the question is how will we make sure the node gets the latest version of record when it comes live?

***Solution:***

The one mechanism followed is repairing node while performing read operations. Since at that point, we can read data from multiple nodes to perform a comparison and find nodes that have stale data. This is called **Read Repair**. Itis a mechanism that solves the problem of stale data in a distributed database system.

***Explanation:***

Based on the quorum, the system reads data from multiple nodes. For example for Quorum=2, the system reads data from one node and digest of the data from the second node.

When a client reads data from a replica, the replica checks the data's timestamp and compares it with the timestamps of the other replicas. If the timestamp is older than the timestamps of other replicas, the replica will update its own data with the more recent version. This process is called "read repair" and it ensures that the data stored on each replica is the most up-to-date version.

It's worth noting that read repair operation can be quite costly in terms of resource utilization as it need to check each replica and update it, hence it is also a trade off in terms of consistency vs performance.

**Real world application:**

Apache Cassandra uses Read Repair strategy.

<https://cassandra.apache.org/doc/latest/cassandra/operating/read_repair.html#:~:text=Cassandra%20uses%20a%20blocking%20read,to%20a%20minority%20of%20replicas>

1. What are Vector clocks?
2. What are the Monotonic Reads?
3. Difference between Sharding and Partitioning?

**Answer.**

1. Different types of NoSQL database?

**Answer.**

**Document databases:** These databases store data in the form of documents, such as JSON. Each document is a self-contained unit of data that contains all the information needed to understand the data. Examples include MongoDB and Couchbase.

**Key-value databases**: These databases store data as key-value pairs, where each key is associated with a value. The key is used to retrieve the value, and the value can be any type of data. Examples include Riak and Redis.

**Column-family databases:** These databases store data in a column-family format, where data is organized into columns rather than rows. Column-family databases are optimized for reading and writing large amounts of data. Examples include Apache Cassandra and HBase.

**Graph databases:** These databases store data in the form of nodes and edges, and are optimized for handling data with complex relationships. This type of databases is used to represent graph data with many relations between elements. Examples include Neo4j, ArangoDB**.**

**Time-series databases:** These databases are designed to handle large amounts of time-stamped data efficiently and used for monitoring, IoT and financial data applications. Examples include InfluxDB, Prometheus.

1. Practices to scale SQL databases?

**Answer.**

**Caching**

**Replication**

**Sharding**

1. Practices to scale NoSQL databases?
2. What is the difference between API Gateway and Load Balancer ?
3. Difference between Rate Limiting and Load Shedding?

**Answer.**

Rate limiting refers to the process of limiting the number of requests that can be made to a system in a specified time period. This helps to prevent a sudden surge of requests from overwhelming the system, which can cause performance issues or even crashes. For example, an online shopping website may limit the number of requests that a single user can make in a minute to prevent scraping and protect their servers from overload.

Load shedding, on the other hand, is a technique used to distribute the load of incoming requests across multiple servers or resources. This helps to ensure that the system remains responsive even under heavy load. Load shedding is often used in conjunction with load balancing techniques to ensure that requests are distributed evenly across multiple servers, helping to avoid overloading a single server. For example, in a cloud computing environment, incoming requests may be automatically redirected to different servers, depending on the load on each server.

In summary, rate limiting is focused on controlling the number of requests that can be made to a system, while load shedding is focused on distributing the load of incoming requests to prevent a single resource from becoming overwhelmed.

1. What is the difference between polling, Long polling
2. What is websocket ?
3. What is server-side event and where is it useful?

**Answer.**

Server-side events (SSEs) is a technology that allows a web server to push data to a web page in real-time. This is in contrast to the traditional request-response model where the web page must constantly poll the server to check for new data.

The main advantage of SSEs is that they allow for real-time updates without the need for constant polling. This can greatly reduce the load on the server and improve the user experience.

SSEs work by establishing a long-lived HTTP connection between the client and the server. The client sends an HTTP request to the server to open the connection, and the server keeps the connection open and sends events as they occur. The client can then receive and process the events as they arrive.

SSEs are useful in a variety of applications, including:

Real-time notifications: A web application can use SSEs to send notifications to users in real-time, such as new messages, friend requests, or updates to their profile.

Real-time analytics: A web application can use SSEs to send real-time analytics data to the client, such as website traffic, user engagement, or sales data.

Real-time collaboration: A web application can use SSEs to enable real-time collaboration between users, such as in a shared document editing application.

Gaming and other real-time applications: SSEs can be used to send real-time updates to users, such as the current state of a game, the position of players, or the status of other real-time events.

It's worth noting that SSEs are not supported by all browsers, and it's best to use a fallback mechanism such as long polling or web sockets in case SSEs are not supported by the client's browser.

1. What is
2. GeoHash and

**Answer.**

GeoHash is a hierarchical spatial data structure that can be used to represent a geographic location as a string of characters. The idea behind GeoHash is to divide the Earth's surface into a grid of cells, where each cell represents a certain area on the Earth's surface. By encoding a location's latitude and longitude as a GeoHash string, you can use the string to easily compare, sort, and search for locations within a certain area.

In conclusion, GeoHash provides a simple and efficient way to encode and search for locations in a geographic database.

Here's an example to help illustrate the concept:

Suppose you have a database of restaurants, and you want to be able to search for restaurants within a certain area, say, a 10-mile radius from a given location. To do this, you can use GeoHash encoding.

First, you'll need to encode the latitude and longitude of each restaurant in your database as a GeoHash string. The length of the GeoHash string determines the precision of the encoding. The longer the string, the smaller the area each cell represents on the Earth's surface.

For example, let's say you have a restaurant located at (37.788022, -122.399797). If you encode this location using a GeoHash string with a precision of 6 characters, the GeoHash string would be "9q8yyz".

Next, you can use the GeoHash string to search for restaurants within a certain area. To do this, you can look for restaurants whose GeoHash strings share a prefix with the GeoHash string of the target location. In this case, you can find all restaurants whose GeoHash strings start with "9q8yy". These restaurants would be located within the same cell as the target location, and therefore, within a 10-mile radius of the target location.

***In more simpler words….***

Let's suppose we have three restaurants: "Pizza Palace", "Burger Barn", and "Taco Temple". We want to find all the restaurants within a certain radius from a location, for example, within a 5 kilometer radius from our current location.

let's say the GeoHash for "Pizza Palace" is "gbsuv7zqz", the GeoHash for "Burger Barn" is "gbsuv7zqm", and the GeoHash for "Taco Temple" is "gbsuv7zpk". Since the first five characters of the GeoHash strings for "Pizza Palace" and "Burger Barn" are the same, we know that they are within the 5 kilometer radius from our location. On the other hand, since the first five characters of the GeoHash for "Taco Temple" are different, it is farther away from our location and outside of the 5 kilometer radius.

This way, we can use GeoHash to find all the restaurants within a certain radius from our location without having to calculate the exact distance between our location and each restaurant.

1. QuadTree

**Answer.**

GeoHash and QuadTree are data structures that are commonly used to store and retrieve geographic information, such as the location of points, lines, or polygons on a map.

**GeoHash** is a data structure that represents a geographical location as a string of characters. It works by dividing the earth's surface into a grid of cells, and then encoding the grid cell that a point lies in as a series of binary digits. For example, if you want to represent the location of a point in San Francisco, you could encode it as a GeoHash string like "9q8yyzbqzpbp". By converting this string back into a grid cell, you can quickly and efficiently retrieve the point's location.

**QuadTree** is a data structure that represents a two-dimensional space, such as a map, as a tree of nested rectangles. Each node in the tree represents a region of the space, and it can have up to four children, representing the four quadrants of the region. When you add a point to a QuadTree, it recursively splits the region until it finds a leaf node that is small enough to contain the point. By searching the tree, you can quickly find the points that are near a given location.

Both GeoHash and QuadTree are widely used in geospatial data management, including in GIS (Geographic Information Systems) and online map services. For example, a service like Google Maps might use a QuadTree to efficiently retrieve the points of interest near a user's location, and then use GeoHashes to efficiently store and retrieve the details of those points.

1. Difference between Strong vs Eventual consistency?

**Answer.**

In distributed systems, consistency refers to the state of data across all nodes in the system. Strong consistency and eventual consistency are two different types of consistency models that can be used in distributed systems.

Strong consistency: In a strongly consistent system, all nodes in the system have the same view of the data at all times. This means that when a write operation is performed on one node, all other nodes will immediately see the updated data. Strong consistency is often achieved through the use of a central authority or a consensus algorithm, such as Paxos.

***Example:***

Banking systems are a good example of a real-world application that uses strong consistency. In banking systems, it is critical that all nodes in the system have the same view of the data at all times in order to ensure that transactions are processed correctly and that account balances are accurate.

Eventual consistency: In an eventually consistent system, there may be a delay before all nodes have the same view of the data. This means that when a write operation is performed on one node, it may take some time for the updated data to be propagated to all other nodes. Eventual consistency is often achieved through the use of replication and asynchronous updates.

***Example:***

* Social media platforms such as Facebook, Twitter and Instagram use Eventual consistency. In these systems, it is not essential that all nodes in the system have the same view of the data at all times. *For example*, if a user posts a new message, it may take some time for that message to appear on all of the user's friends' timelines.
* Another example is online retail websites like Amazon, which may have multiple data centres and warehouses. The inventory in each warehouse may be updated independently and eventually the information is propagated to all data centres.
* Distributed databases like Cassandra, Riak, and Couchbase, which are designed to be highly available and fault-tolerant use eventual consistency model.
* Cloud services like AWS DynamoDB and Google Cloud Datastore are also designed to use Eventual consistency model.

**Differences:**

*One of the key differences* between the two is that with **Strong consistency**, all nodes must see the same value at the same time, while in **Eventual consistency**, there may be a delay before all nodes see the same value.

Another difference is that **Strong consistency** typically requires a centralized mechanism to coordinate updates, while **Eventual consistency** can often be achieved through a more decentralized approach, such as using a distributed consensus algorithm or by using a technique such as CRDTs.

**Strong consistency** is best for systems that require low-latency and immediate updates, but it can be more complex to implement and may be less fault-tolerant. **Eventual consistency** is best for systems that can tolerate some delay in updates, but it is simpler to implement and can be more fault-tolerant.

1. What is the difference between Forward vs Reverse Proxy?

**Answer.**

**Forward Proxy:**

A forward proxy is a server that acts as an intermediary between client machines and the internet. It sits in front of one or more client machines, and when a client makes a request to access a resource on the internet, the request is first sent to the forward proxy. The forward proxy then sends the request on behalf of the client to the internet, and once the response is received it will be sent back to the client machine. This forward proxy also commonly referred as "proxy server" or simply "proxy".

**Reverse Proxy:**

A reverse proxy acts as an intermediary between the internet and one or more web servers. It sits in front of web servers and when a client makes a request to access a resource on the internet, the request is first sent to the reverse proxy. The reverse proxy then forwards the request to one of the web servers, which processes the request and sends back a response. The reverse proxy receives this response and then sends it back to the client. This allows the reverse proxy to handle tasks such as load balancing, SSL termination, and providing an additional layer of security to the web servers.

**Real world applications:**

**Forward proxy**: **Squid** is a popular open-source forward proxy that is widely used by organizations to control access to the internet. Squid can be configured to block access to certain websites or services, such as social media or streaming sites, in order to improve security and reduce distractions.

**Reverse proxy:** **Nginx** and **Apache** are popular open-source reverse proxy servers that are widely used by websites and web services to improve performance and security. These servers can be used to distribute incoming requests across multiple servers, in order to handle the load and improve responsiveness.

**Forward proxy in Cloud**: **Amazon Web Service** Elastic Load Balancer (AWS ELB) can also be used as a forward proxy for distributing incoming traffic across multiple backend servers.

1. What is Rate Limiter and how is it different from Distributed Rate Limiter? and what are the different Rate Limiting algorithms?

**Answer.**

A Rate limiter is used to control the rate of traffic sent by client or service.

In HTTP world, a rate limiter limits the number of client requests allowed to be sent over a specified period. If the API request count exceeds the threshold defined by the rate limiter, all excess calls are blocked.

**Examples:**

* A user can write no more than 2 posts per second
* You can create a maximum of 10 accounts per day from the same IP address.

**Distributed Rate Limiter:**

A distributed rate limiter, is a rate limiter that operates across multiple nodes in a distributed system. The goal of a distributed rate limiter is to enforce a global rate limit that applies to the entire system, regardless of the origin of the incoming requests. To achieve this, the rate limiter must be able to coordinate across multiple nodes and ensure that the rate of incoming requests is consistent and predictable.

***Difference between Rate Limiter and Distributed Rate Limiter:***

The main difference between a rate limiter and a distributed rate limiter is that a rate limiter operates within a single node, while a distributed rate limiter operates across multiple nodes in a distributed system. This makes a distributed rate limiter more complex and challenging to implement, but it also makes it more flexible and capable of handling larger and more complex systems.

Where should we place Rate Limiter?

**Rate Limiter Algorithms:**

* **Token bucket**
  + This algorithm is simple to implement and easy to understand. It works by maintaining a "bucket" of tokens, with each token representing a request that is allowed to pass through the rate limiter. When a request is received, the algorithm checks the number of tokens in the bucket. If there are enough tokens, the request is allowed through and a token is removed from the bucket. If there are no tokens available, the request is blocked. Tokens are added to the bucket at a fixed rate.
* **Leaking bucket**
  + The Leaky Bucket algorithm is similar to the Token Bucket algorithm, but instead of tokens being added to the bucket at a fixed rate, a fixed number of tokens is added to the bucket at regular intervals. This creates a "leak" in the bucket that gradually empties it over time. As with the Token Bucket algorithm, if there are no tokens available when a request is received, the request is blocked.
* **Fixed window counter** 
  + It limits the number of requests a client can make within a fixed interval of time (often a few minutes). It is similar to sliding window algorithm, but the window size does not change with time, it is always fixed.
* **Sliding window counter**
  + This algorithm keeps track of the number of requests made in a fixed-size "sliding window" of time. When a request is received, the algorithm checks the number of requests made in the current window. If the number of requests is within the allowed limit, the request is allowed through. If not, the request is blocked.
* **Smooth bursting**
  + It allows bursts of requests to be handled in a controlled manner. It works by using two different limits: an average rate limit (the number of requests per second that can be made over a long period of time), and a burst limit (the number of requests that can be made over a short period of time, such as a few seconds).
* **IP based rate limiting**
  + It limits the number of requests a client can make based on their IP address. This is easy to implement and effective for preventing DDoS attacks and scrapping bots.

**Real world examples:**

* **Cloudflare:** Cloudflare is a content delivery network (CDN) and DDoS protection provider that uses rate limiting to help protect its customers from DDoS attacks.
* **AWS:** Amazon Web Services (AWS) uses rate limiting in many of its services, such as AWS Elasticsearch and AWS AppSync, to ensure that customers do not exceed their allotted usage.
* **GitHub:** GitHub use rate limiting to control the rate at which authenticated and unauthenticated users can make requests to their API
* **Stripe:** Stripe is a payment processing platform that uses rate limiting to help prevent fraud and abuse of its API.
* **Twilio:** Twilio, a cloud communication platform uses rate limiting to ensure that customers do not send too many messages or make too many API requests within a certain period of time.
* **Google Maps API:** Google Maps API uses rate limiting to control the rate at which requests can be made to its API, helping to ensure that the service remains available to legitimate users.
* **Instagram:** Instagram uses rate limiting to help prevent automated scripts (bots) from scraping data from its website at too high a rate, which would slow down the site for other users.

1. What is the need of *Consensus* in distributed systems?

**Answer.**

Consensus is important in distributed systems because it allows multiple nodes to agree on a single value or state, even in the presence of node failures or network partitions. In a distributed system, multiple nodes may hold copies of the same data, and it's important to ensure that all nodes agree on the latest state of the data, to prevent inconsistencies and data corruption.

***For example***, consider a cluster of 5 nodes that are used to store and manage customer orders. If a customer places an order on node 1, it's important that all other nodes in the cluster agree on the latest state of the customer order. Without consensus, it's possible that one node might have an outdated copy of the customer order, or that two nodes might have different versions of the customer order, which would result in inconsistent data and customer confusion.

To ensure that all nodes in the cluster agree on the latest state of the customer order, a consensus algorithm can be used. A consensus algorithm ensures that, if a node writes a new value to the customer order, all other nodes in the cluster will eventually learn about the new value and update their own copies of the customer order. This ensures that all nodes have a consistent and up-to-date view of the customer order.

1. What are different consensus algorithms?

**Answer.**

**Two-Phase Commit (2PC):** This is a classic consensus algorithm that is used in many databases and other systems. It works by having a coordinator node that coordinates the commit of a transaction across multiple nodes. For example, a distributed database could use 2PC to ensure that all nodes agree on the state of a transaction before it is committed.

**Paxos:** Paxos is a family of consensus algorithms that are used in many distributed systems, including Google's Chubby lock service and Apache Cassandra. Paxos works by having nodes vote on proposals, and once a proposal has been agreed upon by a majority of nodes, it is committed.

**Raft:** Raft is a consensus algorithm that is used in many distributed systems, including Apache ZooKeeper and Etcd. Raft is designed to be easy to understand and implement, and it works by having nodes vote for leaders and agree on the state of the system.

**Byzantine Fault Tolerance (BFT):** BFT is a class of consensus algorithms that are designed to work even in the presence of Byzantine failures, which occur when nodes behave arbitrarily or maliciously. BFT algorithms use cryptographic techniques to ensure that all nodes agree on the state of the system despite the presence of Byzantine failures.

**Proof of Work (PoW) and Proof of Stake (PoS):** PoW and PoS are consensus algorithms that are used in blockchain systems, such as Bitcoin and Ethereum. PoW and PoS work by requiring nodes to perform computationally intensive work or to hold a certain amount of a cryptocurrency in order to participate in consensus.

1. What is Paxos ?

**Answer.**

The Paxos algorithm is a distributed consensus algorithm used to ensure that a set of nodes in a cluster agree on a value. It's designed to work even in the presence of node failures.

Paxos defines several different roles which must cooperate to achieve consensus; they interact to collectively agree on a proposed value. The three roles are:

* Proposers, who receive requests (values) from clients and try to convince acceptors to accept the value they propose.
* Acceptors, who accept certain proposed values from proposers and let proposers know whether a different value was accepted. A response from an acceptor represents a vote for a particular proposal.
* Learners, who announce the outcome to all participating nodes.

Here's an example of how Paxos might work in a cluster with 5 nodes:

* A node initiates a proposal for a value, let's say "42".
* The proposing node sends a "prepare" message to all other nodes in the cluster, asking them to agree to this value.
* The other nodes in the cluster receive the "prepare" message and send back a "promise" message indicating their agreement to the proposal.
* The proposing node then sends a "accept" message to all nodes in the cluster, indicating that it has received enough "promise" messages and is now proposing the value "42".
* The other nodes in the cluster receive the "accept" message and send back an "acknowledge" message indicating their agreement to the value.
* Once the proposing node has received enough "acknowledge" messages, it considers the value "42" to be agreed upon by the cluster and records it.

This process ensures that all nodes in the cluster agree on the value and that the value is recorded even if some nodes fail. It also ensures that the value agreed upon is the most recent value that was proposed.

1. What is Operational Transformation?

**Answer.**

**Problem:**

You may have used Google Docs, which allows multiple people to edit or write a document simultaneously. This is known as **collaborative editing**. Handling concurrent editing in a multi-user environment can be difficult because if multiple people are updating the same document at the same time, it becomes challenging to determine the order of the changes and how the final document should appear.

Let’s take an example:

Starting Client’s state: ABCD

Starting Server’s state: ABCD

Now, let’s say, Client enters X in between C and D , the operation would look something like this :

* insert(X,3) //where 3 is the position where x is going to be added (0=A, 1=B, 2=C ..)

And at the same time, Server deletes B, the operation would be:

* delete(B,1)

What actually should happen is that the client and server should both end with ACXD but in reality, client ends with ACDX.

Starting Document State -> ABCD  
"Insert 'X'" operation at offset 3 [local] -> ABCXD  
"Delete 'B'" operation at offset 1 [remote] -> ACXD

but the server ends with ACDX.

Starting Document State -> ABCD  
"Delete 'B'" operation at offset 1 [local] -> ACD  
"Insert 'X'" operation at offset 3 [remote] -> ACDX

Of course, ACXD!= ACDX and the document which is shared now is in wrong state.

**Explanation:**

A short overview of how OT works:

* Every change (insertion or deletion) is represented as an **operation**. An operation can be applied to the current document which results into a new document state.
* To handle concurrent operations, we use the **transform** function that takes two operations that have been applied to the same document state (but on different clients) and computes a new operation that can be applied after the second operation and that preserves the first operation’s intended change.

Let’s apply Operational Transformation in the original example.

If we apply OT, Client will see :

Starting Document State -> ABCD  
"Insert 'X'" operation at offset 3 [local] -> ABCXD  
"Delete 'B'" operation at offset 1 [transformed] -> ACXD

and Server will see :

Starting Document State -> ABCD  
"Delete 'B'" operation at offset 1 [local] -> ACD  
"Insert 'X'" operation at offset 2 [transformed] -> ACXD //Transform function would add it in the new (3 - 1 = 2) position

**Real world application:**

All collaborative editors like Google Docs.

1. What is CRDT?

**Answer.**

CRDT, or **Conflict-free Replicated Data Types**, is a technique used to ensure consistency and resolve conflicts in distributed systems. It is a way of designing data structures that can be replicated across multiple nodes in a distributed system, and automatically merge different copies of the same data, ensuring that all copies are the same, regardless of the order in which changes were made.

A CRDT is a data structure that has three properties:

Convergence: All copies of the data will eventually converge to the same value, regardless of the order in which changes were made.

Commutativity: The order of applying operations does not affect the final result.

Idempotence: Applying the same operation multiple times has the same effect as applying it once.

There are different types of CRDTs, but one of the simplest is the grow-only counter. A grow-only counter is a counter that can only be incremented, and not decremented. The counter is replicated across multiple nodes, and each node can increment the counter independently.

For example, imagine that you have a website with a "like" button. The website is running on multiple servers, and each server has a copy of the counter. When a user clicks the "like" button, the request is sent to one of the servers, which increments the counter. However, the user may have clicked the button at the same time as another user, and the request may have been sent to a different server. The two servers will have different copies of the counter.

The problem is that the two servers will have different values for the counter, and the website will show different values for the number of likes on different pages. However, with a grow-only counter, the two servers will merge the counters, and the counter will eventually converge to the same value on all servers, regardless of the order in which the increments were made.

1. Difference between CRDT and OT?

**Answer.**

CRDT (Conflict-free Replicated Data Types) and OT (Operational Transformation) are both techniques used to ensure consistency and resolve conflicts in distributed systems. However, they work in slightly different ways.

* CRDTs are a class of data structures that are specifically designed to be replicated across multiple nodes in a distributed system, while OT is a technique used to ensure consistency when multiple users are editing the same data simultaneously.
* CRDTs automatically resolve conflicts without the need for a central authority or coordination between nodes, while OT requires a central authority or coordination between nodes.
* CRDTs have three properties: Convergence, Commutativity, and Idempotence, while OT works by transforming operations on the data so that they can be applied in any order without conflicts.
* CRDTs are used to store data in a distributed system and automatically merge different copies of the same data, while OT can be used in more complex data models than CRDTs
* CRDTs work on the data level, automatically merging data and ensuring consistency, while OT works on operation level, transforming operations to avoid conflicts and ensure consistency.
* CRDTs are best for systems with high read-to-write ratios, while OT is better for systems with high write-to-read ratios.
* CRDTs are best for systems with low latency, while OT can handle high latency systems.

1. What is Raft consensus Algorithm?

**Answer.**

Raft is a distributed consensus algorithm that is designed to be easy to understand, implement, and reason about. It is based on the idea of a *leader* that is responsible for committing log entries to the distributed log.

It was created as an alternative to Paxos, which is notoriously difficult to understand.

The leader is responsible for replicating log entries to a majority of the nodes in the cluster. If a leader fails or becomes unavailable, the other nodes in the cluster will hold an election to choose a new leader. Once a new leader is elected, it will take over the responsibilities of the previous leader and continue to commit log entries to the distributed log.

Please check this awesome visualization around to get more understanding around the same:

<http://thesecretlivesofdata.com/raft/>

Another great article based on “[In Search of An Understandable Consensus Algorithm](https://www.usenix.org/system/files/conference/atc14/atc14-paper-ongaro.pdf) by Diego Ongaro and John Ousterhout”:

<https://www.freecodecamp.org/news/in-search-of-an-understandable-consensus-algorithm-a-summary-4bc294c97e0d/>

**Real world application implementation:**

CockroachDB is a distributed SQL database that is designed to be resilient to failures and scale horizontally. It uses the Raft algorithm to ensure that all nodes in the cluster agree on the state of the data and to handle failures in the cluster.

Etcd is a distributed key-value store that is used for storing and replicating configuration data across a cluster of servers. It uses the Raft algorithm to ensure that the data is consistent across all nodes in the cluster and to handle failures in the cluster.

Apache Kafka distributed message broker also uses the Raft algorithm to ensure that messages are replicated and persisted to a majority of the nodes in the cluster. This ensures that the messages are durable and can be retrieved by consumers even if some nodes in the cluster fail.

1. What’s the difference between Authentication and Authorization?

**Answer.**

Authentication and authorization are two related but distinct concepts in computer security.

Authentication is the process of verifying that a person or system is who or what it claims to be. For example, when you enter your username and password on a website, the website is using authentication to confirm that you are who you claim to be.

Authorization is the process of determining what actions a person or system is allowed to perform. For example, after you have been authenticated, the website may use authorization to determine what pages you are allowed to access based on your role or user level.

In simple terms, authentication is about proving that you are who you say you are, while authorization is about what you are allowed to do with that identity.

For example, when you log in to your bank account, you have to enter your username and password, the bank will check whether it is correct or not. This is the authentication process. Once you are authenticated, the bank will check your account level, what are the rights you have, what you are allowed to do with the account. This is the authorization process.

1. What is a Backpressure?

**Answer.**

When a system is producing data faster than it can be consumed, the system can become overloaded and start to experience performance issues. Backpressure helps to prevent this by slowing down the rate at which data is produced. This can be done by implementing flow control mechanisms, such as buffering, rate limiting, and prioritization.

Backpressure is a mechanism that helps control the flow of data in a system, typically by regulating the rate at which data is produced and consumed. The goal is to prevent a producer from producing data faster than a consumer can consume it, which can lead to a backlog of data and system overload.

An example of backpressure in a data pipeline is when a pipeline stage that consumes data is running slower than the stage that produces data. The consuming stage will buffer incoming data until it can process it, and when the buffer reaches a certain size, it sends a signal to the producing stage to slow down its data production rate. This helps to ensure that the pipeline as a whole run smoothly and avoids overloading the system.

Another way to implement backpressure is by using a message queue. When a consumer is busy or slow, the message queue will buffer messages. And when the buffer reaches its limit, the producer will stop producing messages, this way the queue can avoid from overloading.

1. How to perform API retries?

**Answer.**

Novice way: Retry 3 times

Exponential backoff

Exponential backoff + Jitter

1. Idempotency in API, Stream …
2. How to scale API to handle million calls?

**Answer.**

1. **API Gateways**

API gateways can help you scale your API by reducing the number of API calls and distributing traffic across multiple API endpoints.

1. **Monitor usage and Throttle Requests**

It is important to monitor API usage in order to better understand how the API is being used and to ensure that the API is not being overloaded.

1. **Implement Caching**

Caching can help reduce the load on your API by reducing the number of requests that need to be processed.

1. **Use Asynchronous Processing**

Asynchronous processing can improve the performance of an API by allowing requests to be processed in the background.

1. **Use Multiple Servers**

By implementing a load balancer and multiple servers, you can scale your API to handle a greater number of requests.