Topics: ❑ Load Balancer ❑ Zookeeper ❑ UUID ❑ Consensus ❑ Paxos ❑ Locks ❑ Mutex ❑ Semaphore ❑ Distributed Locks ❑ Long-Polling vs WebSockets vs Server-Sent Events ❑ Event Driven System ❑ ELK stack	❑ CAP theorem ❑ Eventual Consistency ❑ Consistent Hashing ❑ Gossip ❑ Replication factor ❑ Rebalancing ❑ HBase ❑ Map reduce ❑ Cache ❑ Redis ❑ NoSQL ❑ Cassandra ❑ Indexes ❑ Proxies ❑ Sharding or Data partitioning	
Practise: ❑ Bloom Filter ❑ Count Min sketch ❑ API Rate Limiter ❑ Random key generator ❑ URL Shortener	❑ Facebook ❑ Twitter ❑ Whats app ❑ Netflix / Youtube ❑ Amazon	❑ Uber ❑ Google Drive / DropBox ❑ Autocomplete ❑ Google Crawler ❑ Search Engine ❑ Resume Parser

Distributed Systems:

- Key characteristics of a distributed system include Scalability, Reliability, Availability, Efficiency, and Manageability.

Scalability:

- Scalability is the capability of a system, process, or a network to grow and manage increased demand.
- A system may have to scale because of many reasons like increased data volume or increased amount of work, e.g., number of transactions. A scalable system would like to achieve this scaling without performance loss.

Horizontal vs Vertical Scaling:

- Horizontal scaling means that you scale by adding more servers into your pool of resources whereas Vertical scaling means that you scale by adding more power (CPU, RAM, Storage, etc.) to an existing server.
- The ideal system increases capacity linearly with adding hardware. In such a system, if you have one machine and add another, your capacity would double. If you had three and you add another, your capacity would increase by 33%. Let's call this horizontal scalability.
- Good examples of horizontal scaling are Cassandra and MongoDB as they both provide an easy way to scale horizontally by adding more machines to meet growing needs.
- Similarly, a good example of vertical scaling is MySQL as it allows for an easy way to scale vertically by switching from smaller to bigger machines.

Reliability:

- By definition, reliability is the probability a system will fail in a given period. In simple terms, a distributed system is considered reliable if it keeps delivering its services even when one or several of its software or hardware components fail.
- Obviously, redundancy has a cost and a reliable system has to pay that to achieve such resilience for services by eliminating every single point of failure.

Availability:

- By definition, availability is the time a system remains operational to perform its required function in a specific period. It is a simple measure of the percentage of time that a system, service, or a machine remains operational under normal conditions.
- Availability takes into account maintainability, repair time, spares availability, and other logistics considerations.

Reliability vs Availability:

- If a system is reliable, it is available. However, if it is available, it is not necessarily reliable. In other words, high reliability contributes to high availability, but it is possible to achieve a high availability even with an unreliable product by minimizing repair time and ensuring that spares are always available when they are needed.

Efficiency:

Two standard measures of its efficiency are

- the response time (or latency) that denotes the delay to obtain the first item and
- the throughput (or bandwidth) which denotes the number of items delivered in a given time unit (e.g., a second).

The two measures correspond to the following unit costs:

- Number of messages globally sent by the nodes of the system regardless of the message size.
- Size of messages representing the volume of data exchanges.

Serviceability or Manageability:

- Another important consideration while designing a distributed system is how easy it is to operate and maintain. Serviceability or manageability is the simplicity and speed with which a system can be repaired or maintained; if the time to fix a failed system increases, then availability will decrease.

Load Balancing:

- It helps to spread the traffic across a cluster of servers to improve responsiveness and availability of applications, websites or databases.
- LB also keeps track of the status of all the resources while distributing requests.
- If a server is not available to take new requests or is not responding or has elevated error rate, LB will stop sending traffic to such a server.

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

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

Benefits:

- Users experience faster, uninterrupted service.
- Service providers experience less downtime and higher throughput.
- Smart load balancers provide benefits like predictive analytics that determine traffic bottlenecks before they happen.
- System administrators experience fewer failed or stressed components.

LB ALgorithms:

- Least connection
- Least response time
- Least Bandwidth
- Round Robin
- Weighted Round Robin
- IP Hash

How does the load balancer choose backend server:

- Load balancers consider two factors before forwarding a request to a backend server.
- They will first ensure that the server they choose is actually responding appropriately to requests and then use a pre-configured algorithm to select one from the set of healthy servers.

Health Checks:

- Load balancers should only forward traffic to “healthy” backend servers.
- To monitor the health of a backend server, “health checks” regularly attempt to connect to backend servers to ensure that servers are listening.
- If a server fails a health check, it is automatically removed from the pool, and traffic will not be forwarded to it until it responds to the health checks again.

Redundant LB’s:

- The load balancer can be a single point of failure; to

Additional Info on LB:

SSL:

overcome this, a second load balancer can be connected to the first to form a cluster.
- Each LB monitors the health of the other and, since both of them are equally capable of serving traffic and failure detection, in the event the main load balancer fails, the second load balancer takes over.

Smart Clients:
- Adding load-balancing functionality into your database (cache, service, etc) client is usually an attractive solution for the developer. Is it attractive because it is the simplest solution? Usually, no. Is it seductive because it is the most robust? Sadly, no. Is it alluring because it'll be easy to reuse? Tragically, no.
- Developers lean towards smart clients because they are developers, and so they are used to writing software to solve their problems, and smart clients are software.
- With that caveat in mind, what is a smart client? It is a client which takes a pool of service hosts and balances load across them, detects downed hosts and avoids sending requests their way (they also have to detect recovered hosts, deal with adding new hosts, etc, making them fun to get working decently and a terror to setup).

- Secure Sockets Layer (SSL) is the standard security technology for establishing an encrypted link between a web server and a browser.
- SSL traffic is often decrypted at the load balancer. When a load balancer decrypts traffic before passing the request on, it is called SSL termination.
- The load balancer saves the web servers from having to expend the extra CPU cycles required for decryption. This improves application performance.
- **Risk**: The traffic between the load balancers and the web servers is no longer encrypted.

In the seven-layer **Open System Interconnection (OSI) model**,
- network firewalls are at levels one to three
    - L1-Physical Wiring,
    - L2-Data Link and
    - L3-Network.
- Meanwhile, load balancing happens between layers four to seven
    - L4-Transport,
    - L5-Session,
    - L6-Presentation and
    - L7-Application.

LB have different capabilities, which include:
- L4 — directs traffic based on data from network and transport layer protocols, such as IP address and TCP port.
- L7 — adds content switching to load balancing. This allows routing decisions based on attributes like HTTP header, uniform resource identifier, SSL session ID and HTML form data.
- GSLB — Global Server Load Balancing extends L4 and L7 capabilities to servers in different geographic locations.

Apache Zookeeper?
- Distributed systems consist with multiple nodes(computers) which communicate and coordinate their actions by message passing.
- Following are some coordinations happens in distributed systems:
    - 1. Leader election
    - 2. Group membership
    - 3. Locking
    - 4. Synchronisation
    - 5. Publisher/Subscriber
- Zookeeper is a highly available, reliable and scalable distributed system coordination service.
- Zookeeper allows distributed processes to coordinate with each other through a shared hierarchical name space of data registers.
- Suppose you have a distributed web server application running on 10 nodes. Say, you want to get total real-time hit count. One way to do this is to write an

Why Zookeeper?
- While it’s possible to design and implement coordination services from scratch, it’s extra work and difficult to debug any problems, race conditions, or deadlocks.
- You could hack together a very simple group membership service relatively easily, but it would require much more work to write it to provide reliability, replication, and scalability. This led to the development and open sourcing of Apache ZooKeeper, an out-of-the box reliable, scalable, and high-performance coordination service for distributed systems.

Features:

1. Locking
To allow for serialised access to a shared resource in your

application which connects to the 10 nodes, gets count from each and present the sum. Alternatively, you can have each web server application write their hit counts to ZooKeeper on regular intervals and then query ZooKeeper to get the count.

How Zookeeper works?

- ZooKeeper follows a simple client-server model where clients are nodes (i.e., machines) that make use of the service, and servers are nodes that provide the service. A collection of ZooKeeper servers forms a ZooKeeper ensemble
- ZooKeeper service uses the quorum model and will only be available if a majority of servers are alive. The servers in the ZooKeeper service must know about each other. Also, the clients should know the list of servers in the ZooKeeper service.
- At any given time, one ZooKeeper client is connected to one ZooKeeper server(via TCP tunnel). Each ZooKeeper server can handle a large number of client connections at the same time. Each client periodically sends pings to the ZooKeeper server it is connected to let it know that it is alive and connected. The ZooKeeper server in question responds with an acknowledgment of the ping, indicating the server is alive as well. When the client doesn't receive an acknowledgment from the server within the specified time, the client connects to another server in the ensemble, and the client session is transparently transferred over to the new ZooKeeper server.
- For the service to be reliable and scalable, it is replicated over a set of machines. ZooKeeper uses a version of the famous "Paxos algorithm", to keep replicas consistent. An in-memory image of the data tree, transaction logs and snapshots are stored in the replicated machines. Since the data is in-memory, it can achieve low latency and high throughput. However, complete replication limits the total size of data that can be managed using ZooKeeper.

Key concepts of Zookeeper:

- Atomic Broadcast:
  - At the heart of ZooKeeper is an atomic messaging system that keeps all of the servers in sync.
- Reliable delivery:
  - If a message, "m", is delivered by one server, it will be eventually delivered by all servers.
- Total order:
  - If a message is delivered before message "b" by one server, "a" will be delivered before "b" by all servers. If "a" and "b" are delivered messages, either "a" will be delivered before

distributed system, you may need to implement distributed mutexes. ZooKeeper provides for an easy way for you to implement them.

2. Synchronization:
Hand in hand with distributed mutexes is the need for synchronising access to shared resources. Whether implementing a producer-consumer queue or a barrier, ZooKeeper provides for a simple interface to implement that.

3. Configuration Management:
You can use ZooKeeper to centrally store and manage the configuration of your distributed system. This means that any new nodes joining will pick up the up-to-date centralised configuration from ZooKeeper as soon as they join the system. This also allows you to centrally change the state of your distributed system by changing the centralised configuration through one of the ZooKeeper clients.

4. Leader election:
Your distributed system may have to deal with the problem of nodes going down, and you may want to implement an automatic fail-over strategy. ZooKeeper provides off-the-shelf support for doing so via leader election.

5. Name Service:
A name service is a service that maps a name to some information associated with that name. For an example DNS service is a name service that maps a domain name to an IP address. In your distributed system, you may want to keep a track of which servers or services are up and running and look up their status by name. ZooKeeper exposes a simple interface to do that. A name service can also be extended to a group membership service by means of which you can obtain information pertaining to the group associated with the entity whose name is being looked up.

Zookeeper use-cases:

- Zookeeper is ideal for the web-scale, mission critical applications. Also ZooKeeper is popular among multi-host, multi-process applications (C, Java bindings) running in data centers. Following are some zookeeper use cases.

1. **Hbase:**
   HBase is the Hadoop database. Its an open-source, distributed, column-oriented store modeled after the Google paper, Bigtable. They use zookeeper for master election, server lease management, bootstrapping, and coordination between servers.

2. **Neo4j:**

"b" or "b" will be delivered before "a".

- Causal order:
  - If a message "b" is sent after a message "a" has been delivered by the sender of "b", "a" must be ordered before "b". If a sender sends "c" after sending "b", "c" must be ordered after "b".

Neo4j is a Graph Database. It's a disk based, ACID compliant transactional storage engine for big graphs and fast graph traversals, using external indicies like Lucene/Solr for global searches. They use ZooKeeper in the Neo4j High Availability components for write-master election, read slave coordination.

Unique ID in distributed systems:

- All enterprise applications handle tons and tons of data and all this data need an ID to make it distinguishable from some other data.
- In relational database we create primary keys for the same purpose.
- Some databases support inbuilt column types (AUTO_INCREMENT/AUTO_NUMBER) for generating a monotonically increasing 64 bit long number.
- While some people prefer to generate ids in their application layer in order to gain more control over the generation and then persist the records using data layer.
- The later approach however requires you to cache the latest generated number and not lose the track of id's already generated IDs (mostly through some kind of persistence) to avoid Primary key collisions.
- Just think of data shards spread across multiple database nodes how would technique 1 make sure the tables in different nodes do not produce same auto_increment number OR imagine a topology where you are running your application on multiple nodes and these different nodes are serving different parts of the system then how would technique 2 fulfill needs of all nodes.

What is Consensus?

The process by which we reach agreement over system state between unreliable machines connected by asynchronous networks.

Quorum:

Quorum is the number if nodes involved in distributed system.

Paxos:

- Distributed systems are implemented for high availability and scalability using many commodity machines.
- Such machines are not reliable entities and these come up and go down quite often.
- In such systems, there is often a need to agree upon "something" i.e. to have consensus. "Something" here depends on the context.

Consensus problem definition — Basic Paxos

- Even if a single value is proposed, the network

Different approaches to generate Unique ID in distributed systems:

UUID:

- One very simple approach would be generate UUID. UUID's are 128 bits hexadecimal numbers which have a very very very low probability of getting generated twice. So one can simply use any UUID generator and use them for the primary keys.
- UUID' s require no coordination between different nodes and can be generated independently. But then remember they are big in size and they don't index well.

Ticket servers:

- This is one of the very famous approaches where you can simply maintain a table to store just the latest generated ID and every time a node asks for ID they make a 'select for update' on this table, update the value with a incremented value and use the selected value as the next ID.
- This approach is resilient and distributed in nature. The ID generation can be separated from the actual data store.
- However there is a risk of Single Point of Failure as all the nodes rely on this table for the next ID and if this service goes down your app may stop functioning properly.
- Also this approach might not be suitable in case where the writes per second are very high because that will overload the Ticket Server and also degrade your app performance.

Twitter Snowflake:

- This approach tackles the problem of SPOF as well as the latency issues. Here the ID is generated as a concatenation of timestamp, node ID and Sequence number.
- The timestamp is the System time measured as number of millisec. 41 bits are allotted to timestamp.
- Node ID can be assigned to any physical node when during its startup and it can be retrieved from a shared cache in the cluster. Node ID can occupy next 10 bits.
- And the Sequence number can be a monotonically increasing 12 bit number.

eventually recognizes it and choose it.
- Once the value is chosen by the network, it cannot be overwritten.
- Nodes don't hear that a value has been chosen, unless it has really been chosen by the network.

Note:
- These are some famous approaches but you can also check out few other distributed ID generator provided by vendors like Zookeeper and Hazelcast.
- Any JavaScript Number supports only 2^{53}-1 as the maximum safe integral value. So it is always safe to send your IDs as Strings instead of numbers.

Paxos Algorithm:
- In this algorithm, there are three components
  - Proposers(P),
  - Acceptors(A)
  - listener(L)
- Each nodes acts as all three.
- Proposers propose values based on client requests like depositing some money.
- Acceptors accept values and when there is majority/quorum, the proposed value is chosen and committed to the log.

Role of Proposer(P) and Acceptor(A)
- Proposer proposes the value and acceptor accepts the value. Let's define initial RPCs to accomplish consensus.
- P calls prepare(v) RPC on all As. Some A will return some values, but lets ignore those for time being.
- Once majority of A return, P can call accept(v) to those A (depending on the return values from 1). It is possible that return values don't lead to a consensus and then P will have to do a new round of prepare(Vnew) again.

Formalizing the algorithm:
- There are two important items to track. One is the proposal number itself and second one is the value of the proposal.
- Proposer sends proposal number n and broadcasts it to all nodes using the prepare method.
- Acceptor looks at it and makes a promise to not accept any lower proposal i.e. number less than n. If acceptor has seen a higher proposal number then it rejects this proposal.
- Also once an acceptor accepts a proposal value, it will only accept the same proposal value from a higher proposal number.
- If majority of acceptors responded back, then Acceptor uses the value v of the highest proposal number that it got back. When majority didn't agree, then the algorithm needs to go back to step 1.
- Proposer now creates an accept message with the number n and the value v and sends it to the acceptors that responded back in step 2.
- Acceptors will accept this proposal number and value, if they haven't responded back to a higher proposal number already.
- The two phase approach of "prepare and accept" ensures that the only value that can win is the one that gets both prepared and accepted. As long as some P can get in its prepare and accept message out on the majority nodes, Basic paxos will return that value.

Locks:
- A lock is an abstract concept. The basic premise is that a lock protects access to some kind of shared resource. If you own a lock then you can access the protected shared resource. If you do not own the lock then you cannot access the shared resource.
- In simple words whatever code is enclosed in lock scope only one thread can enter in that lock scope and other threads have to wait till entered thread completes its work/ executes its work.
- Locks/Monitors ensures the thread safety which are in process that is threads which are generated by an application (Internal threads) it does not have any control over the threads which are coming from outside of an application.
- Mutex ensures the thread safety which are out

So focusing on the replicated log:
- Each node can run an instance of basic paxos corresponding to every index in the log.
- Each node can select multiple indices simultaneously and reach consensus on the log-value for that index.
- A leader election can be useful to reduce the contention on proposed values
- A leader can batch prepare calls for multiple log indices into a single prepare call and thus eliminate most of the prepare calls.
- A system of heartbeat messages might be needed to discover that a leader is not around any more and then someone else needs to become the leader and proposer. The system can't progress while the leader is down.

process that is threads which coming from outside of an application (external threads).

Mutex:

- Mutex is short for MUTual EXclusion. Unless the word is qualified with additional terms such as shared mutex, recursive mutex or read/write mutex then it refers to a type of lockable object that can be owned by exactly one thread at a time. Only the thread that acquired the lock can release the lock on a mutex. When the mutex is locked, any attempt to acquire the lock will fail or block, even if that attempt is done by the same thread.
- Recursive Mutexes is similar to a plain mutex, but one thread may own multiple locks on it at the same time. If a lock on a recursive mutex has been acquired by thread A, then thread A can acquire further locks on the recursive mutex without releasing the locks already held. However, thread B cannot acquire any locks on the recursive mutex until all the locks held by thread A have been released.

Spinlock

- When you use regular locks (mutexes, critical sections etc), operating system puts your thread in the WAIT state and preempts it by scheduling other threads on the same core.
- This has a performance penalty if the wait time is really short, because your thread now has to wait for a preemption to receive CPU time again.
- Spinlocks don't cause preemption but wait in a loop ("spin") till the other core releases the lock.
- This prevents the thread from losing its quantum and continue as soon as the lock gets released. In a loop a thread waits simply ('spins') checks repeatedly until the lock becomes available.
- The lock is a kind of busy waiting, as the threads remains active by not performing a useful task.

Semaphores

- A semaphore is an integer variables for which only two (atomic) operations are defined, the wait and signal operations You take ownership of a semaphore with a wait operation, also referred to as decrementing the semaphore. You release ownership with a signal operation, also referred to as incrementing the semaphore, a post operation.

Difference between Mutex and a Semaphore

- Mutex is locking mechanism used to synchronize access to a resource. Semaphore is signaling mechanism.
- A mutex is used to meet the atomicity requirement. However, it does not impose any ordering. In other words, given two threads, use of a mutex can't specify, which thread will acquire the mutex first, and hence execute the critical section before the other. The only assurance is that if one thread is executing the critical section, the other will be kept out of it.
- On the other hand, a semaphore can be used to impose ordering constraints in your execution. Considering the example, you can block thread T2 just before it executes instruction Y, conditioned on whether T1 is done executing instruction X. This can be done by making T2 wait on the same condition variable that T1 signals, precisely the programming abstraction that the semaphore provides through its wait() and signal() operations.
- Thus, a mutex can only be used to maintain atomicity whereas a semaphore can be used for both ordering and atomicity.
- Mutex can be released only by thread that had acquired it, while you can signal semaphore from any other thread (or process).

OSI Model: (Open System Interconnection):

- Definition: is a conceptual model that characterises and standardised the communication functions of a telecommunication or computing system without regard to its underlying internal structure and technology. Its goal is the interoperability of diverse communication systems with standard protocols.
- In simpler words, to be precise, the OSI model is a tool used by IT professionals to actually model or trace the actual

flow of how data transfers in networks. So, basically, the OSI model is a logical model/representation of how the network systems are supposed to send data (or, communicate) to each other.

TCP/IP Model:

- The TCP/IP model is a concise version of the OSI model. It contains four layers, unlike seven layers in the OSI model.

Advantages:

- It creates a common platform for software developers and hardware manufactures that encourage the creation of networking products that can communicate with each other over the network.
- It helps network administrators by dividing large data exchange process in smaller segments.
- Due to the independence of layers, it prevents changes in one layer from affecting other layers.
- Standardization of network components allows multiple-vendor development.
- It structures very well the functions particular to each layer.
- It reduces complexity and accelerates evolution
- It simplifies teaching and learning.

Layers	Mnemonics
Application	All
Presentation	People
Session	Seem
Transport	To
Network	Need
Data Link	Data
Physical	Processing

4 Layers of TCP/IP Model:

1. Process/Application Layer
2. Host-to-Host/Transport Layer
3. Internet Layer
4. Network Access/Link Layer

4. Network Access Layer:

- This layer corresponds to the combination of Data Link Layer and Physical Layer of the OSI model. It looks out for hardware addressing and the protocols present in this layer allows for the physical transmission of data.

3. Internet Layer:

- This layer parallels the functions of OSI's Network layer. It defines the protocols which are responsible

7 Layers of OSI Model:

7. Application Layer
This is the topmost layer in the seven OSI Layers. This is the layer that the end-user (can be a computer programmer, or a regular PC user) is actually interacting with. This layer allows access to network resources.

6. Presentation Layer
This is the layer in which the operating system operates with the data. Main functions of this layers includes translation, encryption and compression of data. Basically User interacts with Application layer, which sends the data down to Presentation layer.

5. Session Layer
This layer has the job of maintaining proper communication by establishing, managing and terminating sessions between two computers. For example, whenever we visit any website, our computer has to create a session with the web server of that website.

4. Transport Layer
This layer has a very important job. It decides how much information should be sent at a time. So, when you are communicating with a website, this layer will decide how much data you can transfer and receive at a given point of time. Also, this layer provides reliable process to process message delivery and error recovery.

3. Network Layer
The main job of this layer is to move packets from source to destination and provide inter-networking. This is the layer that the routers operate on. Since routers operate at the network level, hence we can say that the IP address is at the network level.

2. Data Link Layer
This layer is responsible for organising bits into frames and ensuring hop to hop delivery. This is the layer on which the Switches operate on. Since routers operate at the network level, hence we can say that the MAC address resides at the data link layer. All the computers in a specific network get plugged inito a switch so that they can communicate with each other.

1. Physical Layer
This is the layer on which the real transmission of data bits takes place through a medium. This layer is, as the name

for logical transmission of data over the entire network. The main protocols residing at this layer are :

- IP – stands for Internet Protocol and it is responsible for delivering packets from the source host to the destination host by looking at the IP addresses in the packet headers. IP has 2 versions:
- IPv4 and IPv6. IPv4 is the one that most of the websites are using currently. But IPv6 is growing as the number of IPv4 addresses are limited in number when compared to the number of users.
- ICMP – stands for Internet Control Message Protocol. It is encapsulated within IP datagrams and is responsible for providing hosts with information about network problems.
- ARP – stands for Address Resolution Protocol. Its job is to find the hardware address of a host from a known IP address. ARP has several types: Reverse ARP, Proxy ARP, Gratuitous ARP and Inverse ARP.

2. Host-to-host layer:

- This layer is analogous to the transport layer of the OSI model. It is responsible for end-to-end communication and error-free delivery of data. It shields the upper-layer applications from the complexities of data. The two main protocols present in this layer are :

- Transmission Control Protocol (TCP) – It is known to provide reliable and error-free communication between end systems. It performs sequencing and segmentation of data. It also has acknowledgment feature and controls the flow of the data through flow control mechanism. It is a very effective protocol but has a lot of overhead due to such features. Increased overhead leads to increased cost.
- User Datagram Protocol (UDP) – On the other hand does not provide any such features. It is the go-to protocol if your application does not require reliable transport as it is very cost-effective. Unlike TCP, which is connection-oriented protocol, UDP is connectionless.

suggests, all the physical stuff that connects the computers together.

1. Process Layer :

- This layer performs the functions of top three layers of the OSI model: Application, Presentation and Session Layer. It is responsible for node-to-node communication and controls user-interface specifications. Some of the protocols present in this layer are: HTTP, HTTPS, FTP, TFTP, Telnet, SSH, SMTP, SNMP, NTP, DNS, DHCP, NFS, X Window, LPD. Have a look at Protocols in Application Layer for some information about these protocols. Protocols other than those present in the linked article are :

- HTTP and HTTPS – HTTP stands for Hypertext transfer protocol. It is used by the World Wide Web to manage communications between web browsers and servers. HTTPS stands for HTTP-Secure. It is a combination of HTTP with SSL(Secure Socket Layer). It is efficient in cases where the browser need to fill out forms, sign in, authenticate and carry out bank transactions.
- SSH – SSH stands for Secure Shell. It is a terminal emulations software similar to Telnet. The reason SSH is more preferred is because of its ability to maintain the encrypted connection. It sets up a secure session over a TCP/IP connection.
- NTP – NTP stands for Network Time Protocol. It is used to synchronize the clocks on our computer to one standard time source. It is very useful in situations like bank transactions. Assume the following situation without the presence of NTP. Suppose you carry out a transaction, where your computer reads the time at 2:30 PM while the server records it at 2:28 PM. The server can crash very badly if it's out of sync.

HTTP, REST, AJAX polling, HTTP long polling, SSE, Websockets:

- There are so many classifications for APIs. But when it comes to web communication, we can identify two significant API types — Web Service APIs (e.g. SOAP, JSON-RPC, XML-RPC, REST) and Websocket APIs.

HTTP:

- HTTP is the underlying communication protocol of World Wide Web. HTTP functions as a request–response protocol in the client–server computing model.
- HTTP/1.1 is the most common version of HTTP used in modern web browsers and servers.
- In comparison to early versions of HTTP, this version

REST:

- The architectural style, REST (REpresentational State Transfer) is by far the most standardized way of structuring the web APIs for requests.
- REST is purely an architectural style based on several principles.
- The APIs adhering to REST principles are called RESTful APIs.

could implement critical performance optimizations and feature enhancements such as persistent and pipelined connections, chunked transfers, new header fields in request/response body etc.
- Among them, the following two headers are very notable, because most of the modern improvements to HTTP rely on these two headers.
    - Keep-Alive header to set policies for long-lived communications between hosts (timeout period and maximum request count to handle per connection)
    - Upgrade header to switch the connection to an enhanced protocol mode such as HTTP/2.0 (h2,h2c) or Websockets (websocket)

HTTP Polling:

- In HTTP Polling, the client polls the server requesting new information by adhering to one of the below mechanism. Polling is used by the vast majority of applications today and most of the times goes with RESTful practices. In practice, HTTP Short Polling is very rarely used and HTTP Long Polling or Periodic Polling is always the choice.

HTTP Short Polling/ AJAX Polling:

- Simpler approach. A lot of requests are processed as they come to server, creating a lot of traffic (uses resources, but frees them as soon as response is sent back).
- Since each connection is only open for a short period of time, many connections can be time-multiplexed.

00:00:00 C-> Is the cake ready?
00:00:01 S-> No, wait.
00:00:01 C-> Is the cake ready?
00:00:02 S-> No, wait.
00:00:02 C-> Is the cake ready?
00:00:03 S-> Yeah. Have some lad.
00:00:03 C-> Is the other cake ready?

HTTP Long Polling:

- One request goes to server and client is waiting for the response to come.
- The server holds the request open until new data is available (it's unresolved and resources are blocked). You are notified with no delay when the server event happens.
- More complex and more server resources used.

12:00 00:00:00 C-> Is the cake ready?
12:00 00:00:03 S-> Yeah. Have some lad.
12:00 00:00:03 C-> Is the other cake ready?

HTTP Periodic Polling:

- There's a predefined time gap between two requests.
- REST APIs use a request/response model where every message from the server is the response to a message from the client.
- In general, RESTful APIs uses HTTP as its transport protocol.
- For such cases, lookups should use GET requests. PUT, POST, and DELETE requests should be used for mutation, creation, and deletion respectively (avoid using GET requests for updating information).

SSE (Server Sent Events/ EventSource):

- SSE connections can only push data to the browser. (communication is carried out from server to browser only, browsers can only subscribe to data updates originated by server, but cannot send any data to the server)

00:00:00 CLIENT-> I need cakes
00:00:02 SERVER-> Have cake-1.
00:00:04 SERVER-> Have cake-2.
00:00:05 CLIENT-> Enough, I'm full.

- Sample applications: Twitter updates, stock quotes, cricket scores, notifications to browser
- Issue #1: Some browsers don't support SSE.
- Issue #2: Maximum number of open connections is limited to 6 or 8 over HTTP/1.1 (based on browser version). If you use HTTP/2, there won't be an issue because one single TCP connection is enough for all requests (thanks to multiplexed support in HTTP/2).

HTTP/2 Server Push:

- A mechanism for a server to proactively push assets (stylesheets, scripts, media) to the client cache in advance
- Sample applications: Social media feeds, single page apps
- Issue #1: Intermediaries (proxies, routers, hosts) can choose not to properly push information to client as intended by the origin server.
- Issue #2: Connections aren't kept open indefinitely. A connection can be closed anytime even when the content pushing process happens. Once closed and opened again, these connection cannot continue from where it left.
- Issue #3: Some browsers/intermediaries don't support Server Push.

WebSockets:

- WebSockets allow both the server and the client to push messages at any time without any relation to a previous request. One notable advantage in using websockets is, almost every browser support websockets.
- WebSocket solves a few issues with HTTP:

This is an improved/managed version of polling.

- You can reduce server consumption by increasing time gap between two requests.
- But if you need to be notified with no delay when the server event happens, this is not a good option.

00:00:00 C-> Is the cake ready?
00:00:01 S-> No, wait.
00:00:03 C-> Is the cake ready?
00:00:04 S-> Yeah. Have some lad.
00:00:06 C-> Is the other cake ready?

HTTP Streaming:

- Client makes an HTTP request, and the server trickles out a response of indefinite length (it's like polling infinitely).HTTP streaming is performant, easy to consume and can be an alternative to websockets.
- Issue: Intermediaries can interrupt the connection (e.g. timeout, intermediaries serving other requests in round-robin manner). In such cases, it cannot guarantee the complete realtimeness.

00:00:00 CLIENT-> I need cakes
00:00:01 SERVER-> Wait for a moment.
00:00:01 SERVER-> Cake-1 is in process.
00:00:02 SERVER-> Have cake-1.
00:00:02 SERVER-> Wait for cake-2.
00:00:03 SERVER-> Cake-2 is in process.
00:00:03 SERVER-> You must be enjoying cake-1.
00:00:04 SERVER-> Have cake-2.
00:00:04 SERVER-> Wait for cake-3.
00:00:05 CLIENT-> Enough, I'm full.

Web Hooks:

- If you want to obtain data from your API on change of data, polling must be the first option that comes to your mind. But when it comes to communication between servers, inefficiencies in polling cost us a lot (on average, 98.5% of polls get wasted).
- Webhooks are the savior for this problem. Here remember that the communication generally happens between servers. First, the sender node registers a callback URL in the receiver node/s in advance. When an event occurs in the sender side, the webhook gets triggered and sends an event object with new data as a HTTP POST request to the receiver node/s using callback URLs registered in each of them.
- The cool thing is, server load for both the sender and receiver nodes can be reduced drastically with webhooks. It ensures the better user experience while your developers can utilize your service endpoints for meaningful things without wasting for polling.
- Sample applications: Notifications when a new user registers or a current user updates an existing profile setting

    - Bi-directional protocol — either client/server can send a message to the other party (In HTTP, the request is always initiated by client and the response is processed by server — making HTTP a uni-directional protocol)
    - Full-duplex communication — client and server can talk to each other independently at the same time.
    - Single TCP connection — After upgrading the HTTP connection in the beginning, client and server communicate over that same TCP connection throughout the lifecycle of Websocket connection.

00:00:00 CLIENT-> I need cakes
00:00:01 SERVER-> Wait for a moment.
00:00:01 CLIENT-> Okay, cool.
00:00:02 SERVER-> Have cake-1.
00:00:02 SERVER-> Wait for cake-2.
00:00:03 CLIENT-> What is this flavor?
00:00:03 SERVER-> Don't you like it?
00:00:04 SERVER-> Have cake-2.
00:00:04 CLIENT-> I like it.
00:00:05 CLIENT-> But this is enough.

- Sample applications: IM/Chat apps, Games, Admin frontends
- Although websockets are said to be supported every browser, there can be exceptions in intermediaries too:
- Unexpected behaviors in intermediaries: If your websocket connections go through proxies/firewalls, you may have noticed that such connections fail all the times. Always use Secure Websockets (WSS) to drastically reduce such failures. This case is nicely explained here: How HTML5 Web Sockets Interact With Proxy Servers and here too: WebSockets, caution required!. So take the caution and get ready to handle them by using WSS and falling back to a supportive protocol.
- Intermediaries that don't support websockets: If for some reason the WebSocket protocol is unavailable, make sure your connection automatically fallback to a suitable long-polling option.

What to use for you next API?

- **HTTP/1.1 vs HTTP/2**: These are transport protocols. Multiplexing capability in HTTP/2 is great, but not supported everywhere yet. In such cases, make sure falling back from HTTP/2 and HTTP/1.1 won't create any mess in your application. For transferring data, HTTP/1.1 is still a great choice.

- **RESTful APIs**: So far, RESTful APIs are okay for web

- Issue: Developers find it difficult to setup webhooks and scale HTTP services.

Tips:

- Proxies and intermediaries are crazy. They will eat up your packets or timeout unexpectedly. Be aware of that, and handle it gracefully.
- Use secured transport protocols (HTTPS or WSS) to handle your communication. Then, intermediaries will have less impact on your connections. And it's secure, you know.
- Developers love webhooks. But make sure you apply it right.
- You really need not to embrace the latest technology always, especially when you create enterprise level applications. Make sure all the infrastructure is supporting the technology you use. And if the infrastructure does not support your technology/architecture, make sure it falls back to a technology that just works everywhere and everything works gracefully with reduced capabilities.

applications. But there are discussions happening on exploring better ways. An example is this concept — "Replace RESTful APIs with JSON-Pure". I like the idea, in fact, JSON is developer-friendly and you can do wonders with it. But you know, these are developing concepts.

- **JSON vs XML**: Use JSON. It's the trend, and so convenient to deal with too.

- **HTTP Polling:** This is old, but still a great choice for dealing with APIs. If your data is changing frequently or in real time, don't use Short Polling, just use websockets, the better technology for real time. Always use Long and Periodic Polling appropriately (with REST principles).

- **HTTP Streaming:** This is good for applications like news/social media feeds, stock/scoreboards, tweets etc. But in practice, people use websockets than HTTP Streaming.

- **HTTP/2 Server Push** is great for sending resources to client in more managed way. But all intermediaries and browsers will not support this. Make sure you gracefully handle such fallbacks.

- **Server-sent** events are a rather new technology which isn't yet supported by all major browsers, so it is not yet an option for a serious enterprise level web application (Yes, I agree that you can trick this technology to work).

- **WebSockets** provide a richer protocol to perform bi-directional, full-duplex communication. Having a two-way channel is more attractive for things like games, messaging apps, collaboration tools, interactive experiences (inc. micro-interactions), and for cases where you need real-time updates in both directions.

- **Webhooks** are different from all above technologies because it solves a very specific problem. If your servers need to communicate frequently and/or bidirectionally, go for websockets. If your servers communicate occasionally, use REST calls. If your servers need to communicate unidirectionally on an event trigger, then go for webhooks. Don't use polling to check changes in data or state, it's a waste.

Databases

CAP Theorem:

- CAP theorem states that it is impossible for a distributed software system to simultaneously provide

Notes:

- We cannot build a general data store that is continually available, sequentially consistent, and

more than two out of three of the following guarantees (CAP): Consistency, Availability, and Partition tolerance. When we design a distributed system, trading off among CAP is almost the first thing we want to consider. CAP theorem says while designing a distributed system we can pick only two of the following three options:

Consistency:

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

Availability:

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

Partition tolerance:

- The system continues to work despite message loss or partial failure. A system that is partition-tolerant can sustain any amount of network failure that doesn't result in a failure of the entire network. Data is sufficiently replicated across combinations of nodes and networks to keep the system up through intermittent outages.

What is Consistent Hashing?

- Consistent hashing is a very useful strategy for distributed caching system and DHTs. It allows us to distribute data across a cluster in such a way that will minimize reorganization when nodes are added or removed. Hence, the caching system will be easier to scale up or scale down.
- In Consistent Hashing, when the hash table is resized (e.g. a new cache host is added to the system), only 'k/n' keys need to be remapped where 'k' is the total number of keys and 'n' is the total number of servers. Recall that in a caching system using the 'mod' as the hash function, all keys need to be remapped.
- In Consistent Hashing, objects are mapped to the same host if possible. When a host is removed from the system, the objects on that host are shared by other hosts; when a new host is added, it takes its share from a few hosts without touching other's shares.

Consistency here means that a read request for an entity made to any of the nodes of the database should return the same data.

Eventual Consistency:

- Whenever we use multiple replicas of a database to store data and let's say a write request comes to one of the replicas. In such a situation, Databases had to tolerant to any partition failures.
- We can only build a system that has any two of these three properties.
- Because, to be consistent, all nodes should see the same set of updates in the same order.
- But if the network suffers a partition, updates in one partition might not make it to the other partitions before a client reads from the out-of-date partition after having read from the up-to-date one.
- The only thing that can be done to cope with this possibility is to stop serving requests from the out-of-date partition, but then the service is no longer 100% available.

Distributed Hash Table:

- DHT is one of the fundamental components used in distributed scalable systems.
- Hash Tables need a key, a value, and a hash function where hash function maps the key to a location where the value is stored.
- Suppose we are designing a distributed caching system. Given 'n' cache servers, an intuitive hash function would be 'key % n'. It is simple and commonly used. But it has two major drawbacks:
    - It is NOT horizontally scalable. Whenever a new cache host is added to the system, all existing mappings are broken. It will be a pain point in maintenance if the caching system contains lots of data. Practically, it becomes difficult to schedule a downtime to update all caching mappings.
    - It may NOT be load balanced, especially for non-uniformly distributed data. In practice, it can be easily assumed that the data will not be distributed uniformly. For the caching system, it translates into some caches becoming hot and saturated while the others idle and are almost empty.
    - In such situations, consistent hashing is a good way to improve the caching system.

How does Consistent Hashing Work?

- As a typical hash function, consistent hashing maps a key to an integer. Suppose the output of the hash function is in the range of [0, 256). Imagine that the integers in the range are placed on a ring such that the values are wrapped around.

Here's how consistent hashing works:

- Given a list of cache servers, hash them to integers in the range.
- To map a key to a server,
- Hash it to a single integer.
- Move clockwise on the ring until finding the first cache it encounters.

discover a strategy to make this write request at one replica reach other replicas so that they all could also write data of the request and become consistent.
- Eventual consistency makes sure that data of each node of the database gets consistent eventually. Time taken by the nodes of the database to get consistent may or may not be defined.

Strong Consistency:
- It says data will get passed on to all the replicas as soon as a write request comes to one of the replicas of the database.
- But during the time these replicas are being updated with new data, response to any subsequent read/write requests by any of the replicas will get delayed as all replicas are busy in keeping each other consistent.

Conclusion:
- Strong Consistency offers up-to-date data but at the cost of high latency.
- While Eventual consistency offers low latency but may reply to read requests with stale data since all nodes of the database may not have the updated data.

- That cache is the one that contains the key. See animation below as an example: key1 maps to cache A; key2 maps to cache C.
- To add a new server, say D, keys that were originally residing at C will be split. Some of them will be shifted to D, while other keys will not be touched.
- To remove a cache or, if a cache fails, say A, all keys that were originally mapped to A will fall into B, and only those keys need to be moved to B; other keys will not be affected.

For load balancing, as we discussed in the beginning, the real data is essentially randomly distributed and thus may not be uniform. It may make the keys on caches unbalanced.

To handle this issue, we add “virtual replicas” for caches. Instead of mapping each cache to a single point on the ring, we map it to multiple points on the ring, i.e. replicas. This way, each cache is associated with multiple portions of the ring.

If the hash function “mixes well,” as the number of replicas increases, the keys will be more balanced.

Gossip:
- Gossip is used to broadcast members' state around the cluster. Part of the information exchanged:
  - Status
  - Health
  - Tokens
  - schema version
  - Addresses
  - data size

Replication factor
- Replication factor describes how many copies of your data exist.
- For each block stored in HDFS, there will be n – 1 duplicated blocks distributed across the cluster. For example, if the replication factor was set to 3 (default value in HDFS) there would be one original block and two replicas.
- For cassandra, it is 5

Rebalancing:
- Cluster rebalancing ensures that each non-virtual node in a cluster manages an equal amount of data.

SQL Vs NOSQL:
- In the world of databases, there are two main types of solutions: SQL and NoSQL (or relational databases and non-relational databases). Both of them differ in the way they were built, the kind of information they store, and the storage method they use.
- Relational databases are structured and have predefined schemas like phone books that store phone numbers and addresses.
- Non-relational databases are unstructured, distributed, and have a dynamic schema like file folders that hold everything from a person’s address and phone number to their Facebook ‘likes’ and online shopping preferences.

SQL:
- Relational databases store data in rows and columns. Each row contains all the information about one entity and each column contains all the separate data points. Some of the most popular relational databases are MySQL, Oracle, MS SQL Server, SQLite, Postgres, and MariaDB.

NoSQL

Following are the most common types of NoSQL:
- Key-Value Stores: Data is stored in an array of key-value pairs. The ‘key’ is an attribute name which is linked to a ‘value’. Well-known key-value stores include Redis, Voldemort, and Dynamo.
- Document Databases: In these databases, data is

High level differences between SQL and NoSQL

- Storage:
  - SQL stores data in tables where each row represents an entity and each column represents a data point about that entity; for example, if we are storing a car entity in a table, different columns could be 'Color', 'Make', 'Model', and so on.
  - NoSQL databases have different data storage models. The main ones are key-value, document, graph, and columnar. We will discuss differences between these databases below.
- Schema:
  - In SQL, each record conforms to a fixed schema, meaning the columns must be decided and chosen before data entry and each row must have data for each column. The schema can be altered later, but it involves modifying the whole database and going offline.
  - In NoSQL, schemas are dynamic. Columns can be added on the fly and each 'row' (or equivalent) doesn't have to contain data for each 'column.'
- Querying:
  - SQL databases use SQL (structured query language) for defining and manipulating the data, which is very powerful.
  - In a NoSQL database, queries are focused on a collection of documents. Sometimes it is also called UnQL (Unstructured Query Language). Different databases have different syntax for using UnQL.
- Scalability:
  - In most common situations, SQL databases are vertically scalable, i.e., by increasing the horsepower (higher Memory, CPU, etc.) of the hardware, which can get very expensive. It is possible to scale a relational database across multiple servers, but this is a challenging and time-consuming process.
  - On the other hand, NoSQL databases are horizontally scalable, meaning we can add more servers easily in our NoSQL database infrastructure to handle a lot of traffic. Any cheap commodity hardware or cloud instances can host NoSQL databases, thus making it a lot more cost-effective than vertical scaling. A lot of NoSQL technologies also distribute data across servers automatically.

Reasons to use NoSQL database

- When all the other components of our application are stored in documents (instead of rows and columns in a table) and these documents are grouped together in collections. Each document can have an entirely different structure. Document databases include the CouchDB and MongoDB.
- Wide-Column Databases: Instead of 'tables,' in columnar databases we have column families, which are containers for rows. Unlike relational databases, we don't need to know all the columns up front and each row doesn't have to have the same number of columns. Columnar databases are best suited for analyzing large datasets - big names include Cassandra and HBase.
- Graph Databases: These databases are used to store data whose relations are best represented in a graph. Data is saved in graph structures with nodes (entities), properties (information about the entities), and lines (connections between the entities). Examples of graph database include Neo4J and InfiniteGraph.

Reliability or ACID Compliance (Atomicity, Consistency, Isolation, Durability):

- The vast majority of relational databases are ACID compliant. So, when it comes to data reliability and safe guarantee of performing transactions, SQL databases are still the better bet.
- Most of the NoSQL solutions sacrifice ACID compliance for performance and scalability.

SQL VS. NoSQL - Which one to use?

- When it comes to database technology, there's no one-size-fits-all solution. That's why many businesses rely on both relational and non-relational databases for different needs. Even as NoSQL databases are gaining popularity for their speed and scalability, there are still situations where a highly structured SQL database may perform better; choosing the right technology hinges on the use case.

Reasons to use SQL database:

Here are a few reasons to choose a SQL database:

- We need to ensure ACID compliance. ACID compliance reduces anomalies and protects the integrity of your database by prescribing exactly how transactions interact with the database. Generally, NoSQL databases sacrifice ACID compliance for scalability and processing speed, but for many e-commerce and financial applications, an ACID-compliant database remains the preferred option.
- Your data is structured and unchanging. If your business is not experiencing massive growth that would require more servers and if you're only working with data that is consistent, then there may be no reason to use a system designed to support a variety of data types and high traffic volume.

fast and seamless, NoSQL databases prevent data from being the bottleneck. Big data is contributing to a large success for NoSQL databases, mainly because it handles data differently than the traditional relational databases. A few popular examples of NoSQL databases are MongoDB, CouchDB, Cassandra, and HBase.

- Storing large volumes of data that often have little to no structure. A NoSQL database sets no limits on the types of data we can store together and allows us to add new types as the need changes. With document-based databases, you can store data in one place without having to define what "types" of data those are in advance.
- Making the most of cloud computing and storage. Cloud-based storage is an excellent cost-saving solution but requires data to be easily spread across multiple servers to scale up. Using commodity (affordable, smaller) hardware on-site or in the cloud saves you the hassle of additional software and NoSQL databases like Cassandra are designed to be scaled across multiple data centers out of the box, without a lot of headaches.
- Rapid development. NoSQL is extremely useful for rapid development as it doesn't need to be prepped ahead of time. If you're working on quick iterations of your system which require making frequent updates to the data structure without a lot of downtime between versions, a relational database will slow you down.

Partitioning Criteria:

a. Key or Hash-based partitioning: Under this scheme, we apply a hash function to some key attributes of the entity we are storing; that yields the partition number. For example, if we have 100 DB servers and our ID is a numeric value that gets incremented by one each time a new record is inserted. In this example, the hash function could be 'ID % 100', which will give us the server number where we can store/read that record. This approach should ensure a uniform allocation of data among servers. The fundamental problem with this approach is that it effectively fixes the total number of DB servers, since adding new servers means changing the hash function which would require redistribution of data and downtime for the service. A workaround for this problem is to use Consistent Hashing.

b. List partitioning: In this scheme, each partition is assigned a list of values, so whenever we want to insert a new record, we will see which partition contains our key and then store it there. For example, we can decide all users living in Iceland, Norway, Sweden, Finland, or Denmark will be stored in a partition for the Nordic countries.

Data partitioning:

- Data partitioning (also known as sharding) is a technique to break up a big database (DB) into many smaller parts. It is the process of splitting up a DB/table across multiple machines to improve the manageability, performance, availability, and load balancing of an application. The justification for data sharding is that, after a certain scale point, it is cheaper and more feasible to scale horizontally by adding more machines than to grow it vertically by adding beefier servers.

Partitioning Methods:

There are many different schemes one could use to decide how to break up an application database into multiple smaller DBs. Below are three of the most popular schemes used by various large scale applications.

a. Horizontal partitioning: In this scheme, we put different rows into different tables. For example, if we are storing different places in a table, we can decide that locations with ZIP codes less than 10000 are stored in one table and places with ZIP codes greater than 10000 are stored in a separate table. This is also called a range based sharding as we are storing different ranges of data in separate tables.
The key problem with this approach is that if the value whose range is used for sharding isn't chosen carefully, then the partitioning scheme will lead to unbalanced servers. In the previous example, splitting location based on their zip codes assumes that places will be evenly distributed across the different zip codes. This assumption is not valid as there will be a lot of places in a thickly populated area like Manhattan as compared to its suburb cities.

b. Vertical Partitioning: In this scheme, we divide our data to store tables related to a specific feature in their own server. For example, if we are building Instagram like application - where we need to store data related to users, photos they upload, and people they follow - we can decide to place user profile information on one DB server, friend lists on another, and photos on a third server.
Vertical partitioning is straightforward to implement and has a low impact on the application. The main problem with this approach is that if our application experiences additional growth, then it may be necessary to further partition a feature specific DB across various servers (e.g. it would not be possible for a single server to handle all the metadata queries for 10 billion photos by 140 million users).

c. Directory Based Partitioning: A loosely coupled approach to work around issues mentioned in the above schemes is to create a lookup service which knows your current partitioning scheme and abstracts it away from the DB access code. So, to find out where

c. Round-robin partitioning: This is a very simple strategy that ensures uniform data distribution. With 'n' partitions, the 'i' tuple is assigned to partition (i mod n).
d. Composite partitioning: Under this scheme, we combine any of the above partitioning schemes to devise a new scheme. For example, first applying a list partitioning scheme and then a hash based partitioning. Consistent hashing could be considered a composite of hash and list partitioning where the hash reduces the key space to a size that can be listed.

Indexes:

- Indexes are well known when it comes to databases. Sooner or later there comes a time when database performance is no longer satisfactory. One of the very first things you should turn to when that happens is database indexing.
- The goal of creating an index on a particular table in a database is to make it faster to search through the table and find the row or rows that we want. Indexes can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access of ordered records.
- Example: A library catalog
    - A library catalog is a register that contains the list of books found in a library. The catalog is organized like a database table generally with four columns: book title, writer, subject, and date of publication. There are usually two such catalogs: one sorted by the book title and one sorted by the writer name. That way, you can either think of a writer you want to read and then look through their books or look up a specific book title you know you want to read in case you don't know the writer's name. These catalogs are like indexes for the database of books. They provide a sorted list of data that is easily searchable by relevant information.
    - Simply saying, an index is a data structure that can be perceived as a table of contents that points us to the location where actual data lives. So when we create an index on a column of a table, we store that column and a pointer to the whole row in the index. Let's assume a table containing a list of books, the following diagram shows how an index on the 'Title' column looks like:

- Just like a traditional relational data store, we can also apply this concept to larger datasets. The trick with indexes is that we must carefully consider how users will access the data. In the case of data sets that are

a particular data entity resides, we query the directory server that holds the mapping between each tuple key to its DB server. This loosely coupled approach means we can perform tasks like adding servers to the DB pool or changing our partitioning scheme without having an impact on the application..

Common Problems of Sharding

On a sharded database there are certain extra constraints on the different operations that can be performed. Most of these constraints are due to the fact that operations across multiple tables or multiple rows in the same table will no longer run on the same server. Below are some of the constraints and additional complexities introduced by sharding:

a. Joins and Denormalization: Performing joins on a database which is running on one server is straightforward, but once a database is partitioned and spread across multiple machines it is often not feasible to perform joins that span database shards. Such joins will not be performance efficient since data has to be compiled from multiple servers. A common workaround for this problem is to denormalize the database so that queries that previously required joins can be performed from a single table. Of course, the service now has to deal with all the perils of denormalization such as data inconsistency.
b. Referential integrity: As we saw that performing a cross-shard query on a partitioned database is not feasible, similarly, trying to enforce data integrity constraints such as foreign keys in a sharded database can be extremely difficult.
Most of RDBMS do not support foreign keys constraints across databases on different database servers. Which means that applications that require referential integrity on sharded databases often have to enforce it in application code. Often in such cases, applications have to run regular SQL jobs to clean up dangling references.
c. Rebalancing: There could be many reasons we have to change our sharding scheme:
The data distribution is not uniform, e.g., there are a lot of places for a particular ZIP code that cannot fit into one database partition.
There is a lot of load on a shard, e.g., there are too many requests being handled by the DB shard dedicated to user photos.
In such cases, either we have to create more DB shards or have to rebalance existing shards, which means the partitioning scheme changed and all existing data moved to new locations. Doing this without incurring downtime is extremely difficult.
Using a scheme like directory based partitioning does make rebalancing a more palatable experience at the cost of increasing the complexity of the system and creating a new single point of failure (i.e. the lookup

many terabytes in size, but have very small payloads (e.g., 1 KB), indexes are a necessity for optimizing data access. Finding a small payload in such a large dataset can be a real challenge, since we can't possibly iterate over that much data in any reasonable time. Furthermore, it is very likely that such a large data set is spread over several physical devices—this means we need some way to find the correct physical location of the desired data. Indexes are the best way to do this.

Proxies:

- A proxy server is an intermediate server between the client and the back-end server. Clients connect to proxy servers to request for a service like a web page, file, connection, etc. In short, a proxy server is a piece of software or hardware that acts as an intermediary for requests from clients seeking resources from other servers.
- Typically, proxies are used to filter requests, log requests, or sometimes transform requests (by adding/removing headers, encrypting/decrypting, or compressing a resource). Another advantage of a proxy server is that its cache can serve a lot of requests. If multiple clients access a particular resource, the proxy server can cache it and serve it to all the clients without going to the remote server.

Caching:

- Load balancing helps you scale horizontally across an ever-increasing number of servers, but caching will enable you to make vastly better use of the resources you already have as well as making otherwise unattainable product requirements feasible.
- Caches take advantage of the locality of reference principle: recently requested data is likely to be requested again.
- They are used in almost every layer of computing: hardware, operating systems, web browsers, web applications, and more.
- A cache is like short-term memory: it has a limited amount of space, but is typically faster than the original data source and contains the most recently accessed items.
- Caches can exist at all levels in architecture, but are often found at the level nearest to the front end where they are implemented to return data quickly without taxing downstream levels.

service/database)

How do Indexes decrease write performance?

- An index can dramatically speed up data retrieval but may itself be large due to the additional keys, which slow down data insertion & update.
- When adding rows or making updates to existing rows for a table with an active index, we not only have to write the data but also have to update the index. This will decrease the write performance.
- This performance degradation applies to all insert, update, and delete operations for the table. For this reason, adding unnecessary indexes on tables should be avoided and indexes that are no longer used should be removed. To reiterate, adding indexes is about improving the performance of search queries.
- If the goal of the database is to provide a data store that is often written to and rarely read from, in that case, decreasing the performance of the more common operation, which is writing, is probably not worth the increase in performance we get from reading.

Proxy Server Types

Proxies can reside on the client's local server or anywhere between the client and the remote servers. Here are a few famous types of proxy servers:

- Open Proxy: An open proxy is a proxy server that is accessible by any Internet user. Generally, a proxy server only allows users within a network group (i.e. a closed proxy) to store and forward Internet services such as DNS or web pages to reduce and control the bandwidth used by the group. With an open proxy, however, any user on the Internet is able to use this forwarding service. There two famous open proxy types:
    - Anonymous Proxy - This proxy reveals its identity as a server but does not disclose the initial IP address. Though this proxy server can be discovered easily it can be beneficial for some users as it hides their IP address.
    - Transparent Proxy – This proxy server again identifies itself, and with the support of HTTP headers, the first IP address can be viewed. The main benefit of using this sort of server is its ability to cache the websites.
- Reverse Proxy: A reverse proxy retrieves resources on behalf of a client from one or more servers. These resources are then returned to the client, appearing as if they originated from the proxy server itself

Application server cache

Content Distribution Network (CDN)

- CDNs are a kind of cache that comes into play for sites serving large amounts of static media. In a typical CDN setup, a request will first ask the CDN for a piece of static media; the CDN will serve that content if it has it locally available. If it isn't available, the CDN will query the back-end servers for the file, cache it locally, and serve it to the requesting user.
- If the system we are building isn't yet large enough to have its own CDN, we can ease a future transition by serving the static media off a separate subdomain (e.g. static.yourservice.com) using a lightweight HTTP server like Nginx, and cut-over the DNS from your servers to a CDN later.

Cache eviction policies:

Following are some of the most common cache eviction policies:

- First In First Out (FIFO): The cache evicts the first block accessed first without any regard to how often or how many times it was accessed before.
- Last In First Out (LIFO): The cache evicts the block accessed most recently first without any regard to how often or how many times it was accessed before.
- Least Recently Used (LRU): Discards the least recently used items first.
- Most Recently Used (MRU): Discards, in contrast to LRU, the most recently used items first.
- Least Frequently Used (LFU): Counts how often an item is needed. Those that are used least often are discarded first.
- Random Replacement (RR): Randomly selects a candidate item and discards it to make space when necessary.

- Placing a cache directly on a request layer node enables the local storage of response data. Each time a request is made to the service, the node will quickly return local cached data if it exists. If it is not in the cache, the requesting node will query the data from disk. The cache on one request layer node could also be located both in memory (which is very fast) and on the node's local disk (faster than going to network storage).
- What happens when you expand this to many nodes? If the request layer is expanded to multiple nodes, it's still quite possible to have each node host its own cache. However, if your load balancer randomly distributes requests across the nodes, the same request will go to different nodes, thus increasing cache misses. Two choices for overcoming this hurdle are global caches and distributed caches.

Cache Invalidation:

- While caching is fantastic, it does require some maintenance for keeping cache coherent with the source of truth (e.g., database). If the data is modified in the database, it should be invalidated in the cache; if not, this can cause inconsistent application behavior.
- Solving this problem is known as cache invalidation; there are three main schemes that are used:
    - Write-through cache: Under this scheme, data is written into the cache and the corresponding database at the same time. The cached data allows for fast retrieval and, since the same data gets written in the permanent storage, we will have complete data consistency between the cache and the storage. Also, this scheme ensures that nothing will get lost in case of a crash, power failure, or other system disruptions. Although, write through minimizes the risk of data loss, since every write operation must be done twice before returning success to the client, this scheme has the disadvantage of higher latency for write operations.
    - Write-around cache: This technique is similar to write through cache, but data is written directly to permanent storage, bypassing the cache. This can reduce the cache being flooded with write operations 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.
    - Write-back cache: Under this scheme, data is written to cache alone and completion is immediately confirmed to the client. The write

	to the permanent storage is done after specified intervals or under certain conditions. This results in low latency and high throughput for write-intensive applications, however, this speed comes with the risk of data loss in case of a crash or other adverse event because the only copy of the written data is in the cache.

Note: Information gathered in this document has been collected from various sources on Internet.
Sources: Educative.io (Grokking System Design), https://medium.com/platform-engineer/web-api-design-35df8167460