**System Design** is about designing a system’s components, architecture, and modules and how data will be flowing from one element to another. In System Design, We focus on solving a problem by specifying all necessary components for a sound running system. System Design is also a part of problem-solving on a larger scale.

5 Important steps to design a reliable system:

1. **Designing Interfaces (GUI, API):** - Making signatures for API Designing (Rest API or 1 GraphQL ) and how many screens are required to complete the Application. The interviewer checks how broadly you are thinking about your application
2. **Capacity Approximation**: We ask the interviewer about scenarios like how much our application is scaled, how many DB Calls per second would be there.
3. **Selection of the right Database and Data Structure**: Choosing the right DB, files, RAM based on our storage and server type requirement is the next step.
4. **High-Level Detailed Design**: show the flow of data from Client to Server and vice-versa using boxes and flow diagrams. You need to provide the number of nodes (hosts) of the server, Load Balancing, Caching, DB persistence, and all other details required to design a better possible solution.
5. **Optimization**: Check for Point of failures, if backup requirements are there, bottleneck conditions.

**Important Factors to be considered while designing a system:**

**Scalability:** It simply means making the system flexible enough to accommodate new users and handle efficiently increased data rate and traffic load. The performance and management of the system and complexity shouldn’t be affected while scaling.

**Throughput:** It is defined as the amount of work done by a machine in a given particular time. Throughput is one of the significant metrics for Network performance. Generally, we measure the throughput of the server how many API Calls it serves in a unit of time. Generally, we measure throughput in GBps, MBps, KBps. How to increase throughput?

* Simple answers can be paying for the good server if higher throughput is needed
* By increasing the capacity of load taken by the server.
* By increasing the speed of work done by the system.

Bandwidth: Maximum data that can be transferred over different Networks is known as Bandwidth.

**Response Time**: Time taken by any API in response to API Call is response time. Let’s say we sent a request to API at t1, and you got a response at t2 that the time difference between request and response is response time.

**Latency**: Latency is an important measure for the performance of the system. It is defined as how long a system takes to transmit data from one point to another point in the system.

* A Network request latency is the time taken by a request sent by the client and server sending the same request’s response.
* Time is taken for reading data from a file on Server.
* If we perform ten types of operations in our system, latency can differ for different functions.

**Availability:** It simply means how fault tolerant a system is. If any failure occurs, then how will the system work? That is how resistant a system is in adverse conditions and how it repairs itself whenever needed.

**Consistency:** When two or more nodes are supposed to share the same data, any node can manipulate data first, and the other node is also trying to access, while node A writes something at t1 time, and node B read before t1, but the data got updated in meanwhile, leading to inconsistency. To avoid inconsistency, we can use Master-Slave architecture, which can become peer to peer in some cases while two replica share data at the same level.

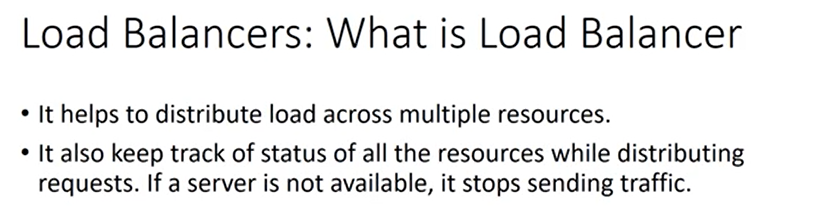
**What is the difference between Weak, Strong, and Eventual Consistency?**

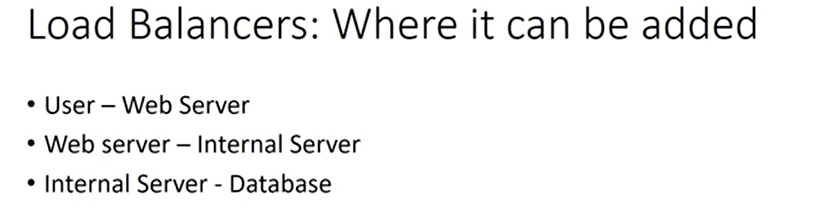
Eventual Consistency: This simply means that the data must be highly available and partially consistent. In case of and overlapping read with an edit, the read might be inconsistent at first but later it gets updated to a consistent value. Here availability is the priority over consistency but none the less consistency is also achieved**.**

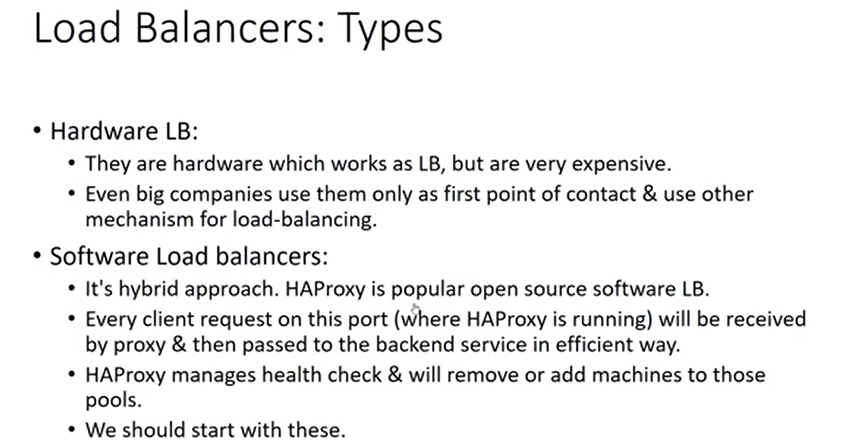
Strong Consistency: Strong Consistency simply means the data must be strongly consistent at all times. All the server nodes across the world should contain the same value as an entity at any point in time. And the only way to implement this behaviour is by locking down the nodes when being updated.

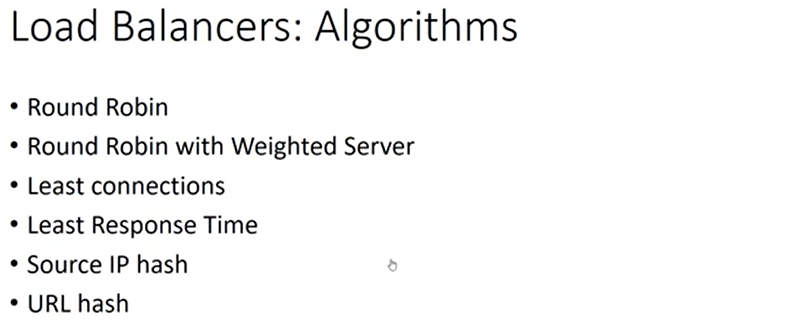
**What are some examples of availability patterns?** H/W redundancy for application servers and DB Master-slave model and replication.

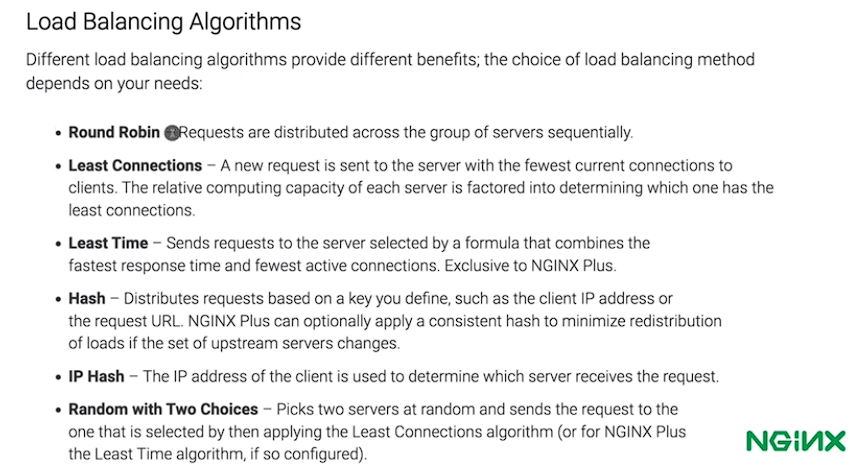
**Load Balancer:**











**Caching/Cache memory:**

Cache is defined as a Hardware or Software component which helps in serving the data which is either frequently requested or is hard to fetch/complex or expensive to compute on.

Cache can be built up on reverse proxy org server which processes the request in absence of cache or can be built up on forward proxy server which distributes or routs the data to the application server.

Cache hit and Cache miss.

Cache Invalidation:

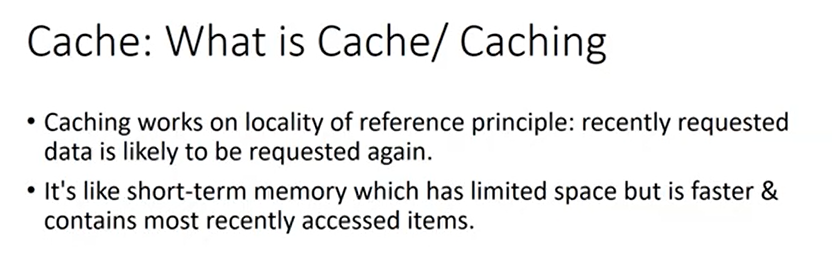
Data stored in cache is volatile and does not exist forever. Cache invalidation techniques are needed when the stored data is changeable depending on the time. The cache’s K-V would be needed to be updated or removed in case that particular data is changed/updated in persistent storage (warming up of cache memory). Hence faulty or invalid data members of cache are needed to either be updated or removed from the cache storage.

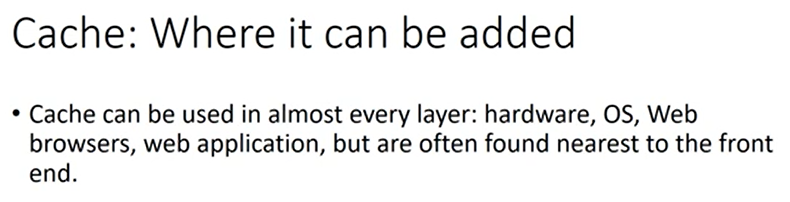
Few Techniques:

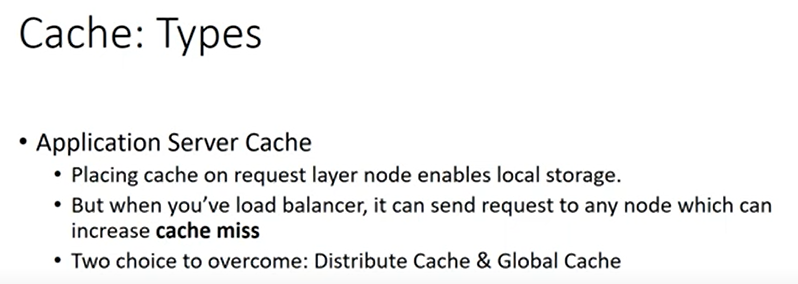
TTL (time of cache expiration) (if TTL is too short, no point of using a cache and if the TTL is too long, value inconsistency might arise) hence deciding on TTL value is a non-deterministic problem and is totally use case dependent.

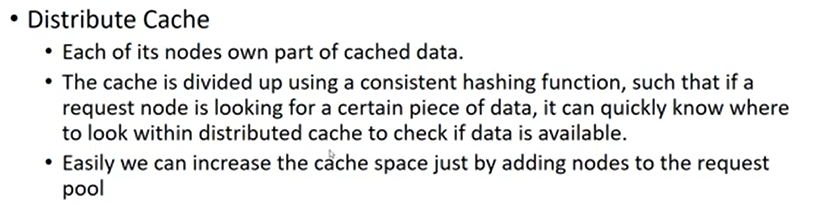
Another way is by taking help of Application server. (This is only possible if cached data is on reverse proxy server). Either application server can check if the write request is on a data object whose key is present in cache and remove it or it can check for the same and update the K-V pair in Persistent storage simultaneously. We can use the Hybrid of TTL and Application layer handling as well.

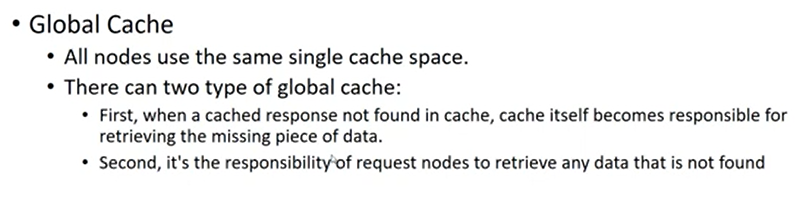
Cache Eviction: If the cache is full, it is needed to be handled in a way that no useful data is evicted and hence eviction policies are used to make space for the new entries.

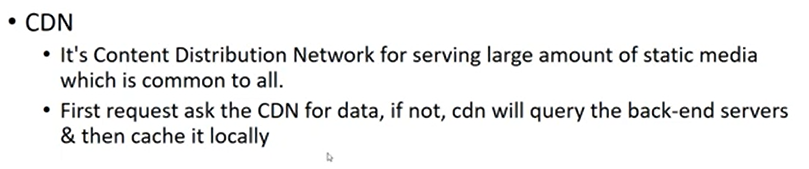


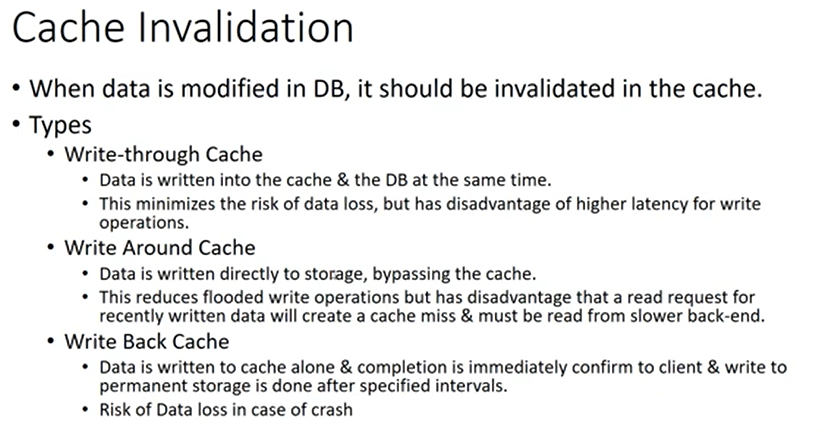


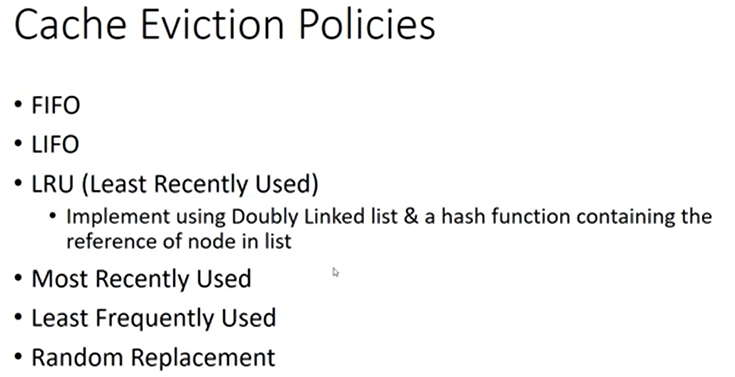










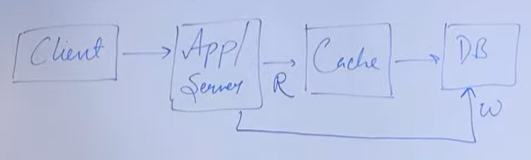


Cache Patterns:

Cache Aside Pattern: (+System availability in case of cache failure [in this case the system might slow down but the availability is still present], -Cache invalidation technique used might be complex)

Read/Write Through cache: (+Good for read heavy applications, -Single point of failure, and extra layer of latency in case of write operation)

Write around Cache:



Write Back Cache: (All the writes are stored in cache and passed to persistent layer in batches and hence it can handle DB failure for some time. –If cache goes down all the writes are lost. This is especially used in write heavy applications).

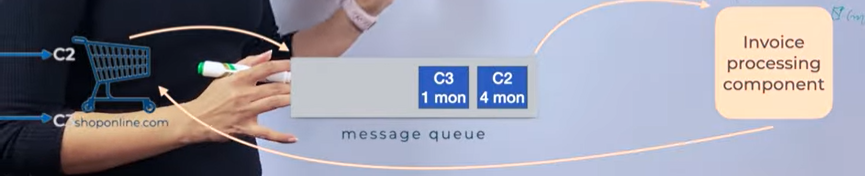


**Message Queue:**

**Synchronous Communication**: Connected (generally 2) components communicated over a dedicated line to exchange messages in a synchronous way. There is no lag at either components for processing the request. Processing is spontaneous and fast and hence the exchange goes sequentially. (Eg: Phone call)

**Asynchronous Communication**: Connected components have a communication lag/defer to process a certain request while not necessarily holding the seeker. Spontaneous response in not expected as the requested process might take some time. Hence in case of async-await there is a function lag to process and utilize the result or output of the requested process while maintaining consistency. (Eg: text and mails.)

Message Queue is a Queue data structure that is utilised to achieve a highly scalable asynchronous communication. Example: (Order Invoice range request by multiple users)



Here whole communication is asynchronous. Request are time intensive and need more time produce the desired result. It provides ease to the system to process multiple requests in a non-simultaneous fashion plus it makes the system request scalable. Here the 2 components can communicating are online store and invoice processing component. Examples: Kafka / Rabbit MQ / Active MQ.

Message could be a data structure to store upcoming requests or it could be a process running on either same machine or different machine that is needed to be contacted for processing the stored requests. This queue contains messages that are nothing but short sized data which just tells what task is needed to be done. Now what are the components that communicate using this queue? These are producers and subscribers. The components that add message on to the queue are producers while the components that carry out the required task or aid in the desired process are called consumers.

**Advantages**: Handle multiple (lot of) requests and decouple the system into multiple producers and consumers according to the load thereby providing scalability. Message queue is also reliable in case if the components lose / faces an unwanted failure, queue still holds the requested attributes to once again start the processing.

**Ordering**: Once the consumer processes the front request and acknowledges the result as required, then only the request is dequeued from the queue.

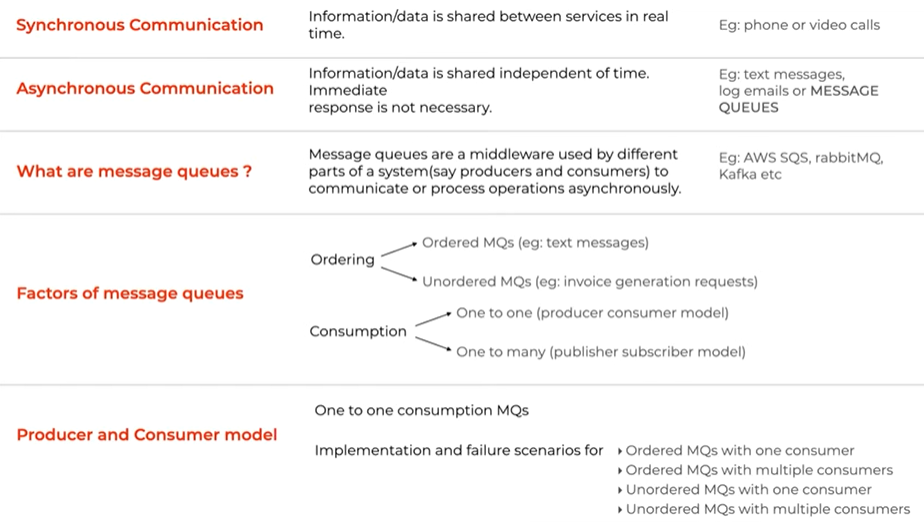
**Ordered Message queues**: Amazon SQS, ordering is imperative and done in FIFO manner. In case if some request is failed to be processed, queue is blocked until the front() message is not processed and acknowledged to the queue. This ensures ordering. Example (Chat application). This happens similarly in case of multiple consumer presence. Functioning gets blocked until the expected consumer do not completes the unfinished request.

**Unordered Message Queue**: Ordering isn’t important here. In case of failure in a request process, the failed message is pushed onto a DEAD Letter Queue (DLQ) and is pushed again into the Message queue for re-processing.

**Message Queue Models**:

Producer and Consumer: Consumption is 1 to 1, One message is consumed only once by one consumer.

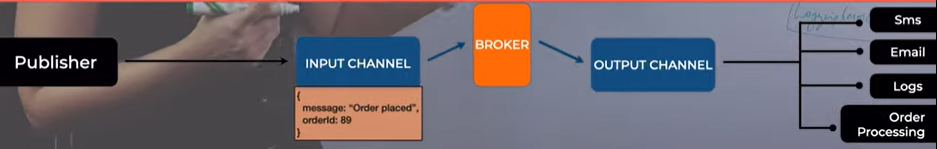
Publish Subscribe: Consumption is 1 to Many: For one message multiple processes have to do something in response to that message.

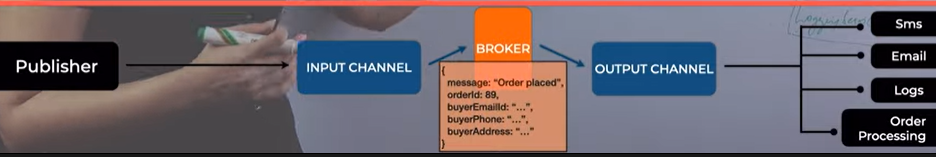


Publish Subscribe is also a pattern for asynchronous communication. Analogy: “Hospital announcements analogy”.

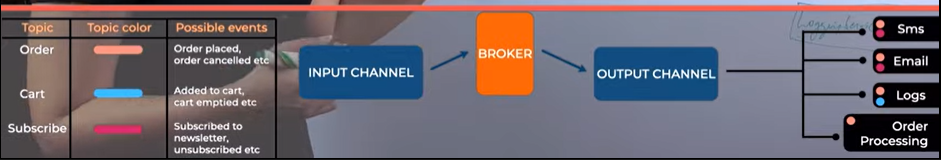
In any system, a component that informs about certain event that has happened is called a Publisher. It pushes the messages informing what event has happened in the system. An Input channels is a physical component where publisher will push all the event messages. Output channel is where the subscribers who reacts to the respective event messages subscribe. Now there could be multiple output channels depending on events concerning certain subscribers (A subscriber need not to be informed about all the published event messages, hence this subscriber would need a separate output channel which will inform it about only the concerned events). Message broker in the middle has to hold on with multiple responsibilities like:

1. Enriching/modifying messages as per the requirement by the subscribers.
2. Message broker can divide the messages into different topics. Relating to our previous analogy, an ER announcement (topic: ER, ER announcement: event message) is needed to be heard by everyone whereas speciality announcement (childcare dept. doctors) are concerned and done only on specific floor(s) (this event messages gets routed onto the concerned output channels).









Publish Subscribe model allows decoupling of components and scaling of the system very easily. It also increases system performance. Ordering here is not decided and consumption is 1 to many. In order to achieve a certain ordering we would need to priority to every event message and push it into a priority queue instance. Repeated and 3rd party generated ill-messages are needed to be taken care for.

**Database Replication:**

Replication means to copy/or to have a copy, in this case- of a Database (DB). This process is done to avoid Single point of failure and data consistency. Main database which has the write authority is called the master / Leader DB and the one that continuously copies the master database and replicates the data from master DB is called slave or secondary DB and they are usually >1 and are used for read operation. How replication helps?

* Fault tolerance: Let us say, some mishap happens with one of the DB’s hardware. With replicated DB available, we can have all the data still available and this can now be as master DB.
* Reduce Latency: because distributed and replicated FS / DB makes the information access fast. A geographically close location will surely help us reduce delivery time as compared to a far located DB on the same network.
* Increase system performance and make it scalable as replica will help out in sorting multi-read operation’s burden.

How replication happens: One could be periodically send master DB snaps to the replicas or using CDC (Change Data capture) slight change in Master DB initiates a change forwarding process to update all replicas.

Replication Lag: When a read and write request occurs at the same time. Let us say if write request happens at T1 and a read request occurs at T2 and the systems replication lag is R, then,

* Everything is fine until T2-T1>R; Till the time T2 secondary DB would be updated as per the write request of T1. But if T2-T1<=R, data consistency is hampered, and dirty read will happen.

This is the challenge faced in a DB replication model. This is resolved by consistency algorithms like “read after write”; Let us say a new write request is issued at T4 on master DB (obviously). Now with this model of “read after write”, master will updated according to the request and send the update changes to all the replicas and wait for their acknowledgement(s). When it receives all the acknowledgements is will inform the request issuer that the change is updated in the database and hence Replication lag is 0 with full data consistency. This is called Synchronous Replication. The downsides here are that system might take a performance hit as the processing of the write request holds onto the other operations. Here in case of replica failure scenario during acknowledgement phase also takes a lot of consideration.

Asynchronous Replication: Primary DB will not wait for the replicas’ acknowledgements and the replica update process is done in the background. When the write request is issued at the master, master will first forward this request to the replica DBs and then will update itself according to the request. Advantage is of lowering the chance of performance hit of DB but there still might be some inconsistency in this system.

Semi-synchronous Replication: When a new write request is issued, master DB will forward the request to all the replicas but will wait for acknowledgement from only one replica. Then the Master DB will also update itself and informs the request issuer about the update.

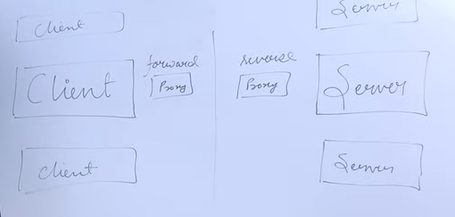
Snapshot vs. Replica: Snapshot is a state of DB at a certain time whereas DB replica is the full copy of the master DB. Snapshot allows the DB system to roll back to a recently saved snapshot but it won’t allow you to reduce latency and scale in case of multi read operation thereby increase performance as the replica do.

**Proxy server:** Analogy (broker (asset) between 2 parties for communication and abstraction for security);

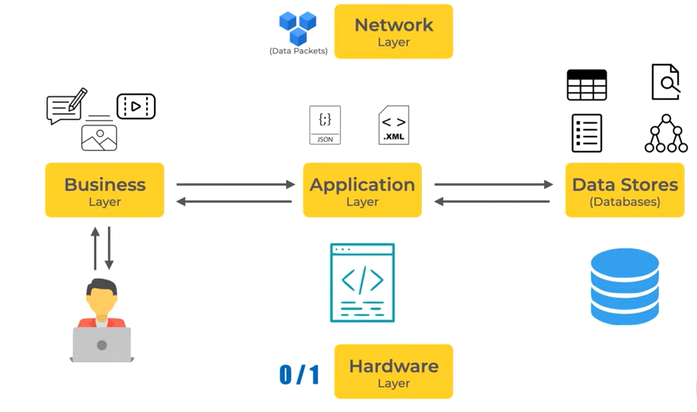
**Forward Proxy**: Think about proxy server as a machine which sits between the client and the request handling server, but on the client side which talk to the server on behalf of the client. Useful for anonymity. This is useful when you have multiple clients and all the traffic is to be monitored and controlled via forward proxy. This can also be used when certain access permissions are to be granted within a general group of requests. (Can also be used for blocking access to certain site or Caching user requesting on the proxy site). In certain scenarios this type of proxy can be used for illegal accesses.

**Reverse Proxy**: When the proxy server sits in between client and the request handling servers but on the server side and client only has the knowledge about the proxy server which in turn distributes the request onto different request handling servers. This setup is majorly used for security purpose (like request and data SSL encryption) and abstracting actual server details and it can also be used as corporation’s Load balancing server. It becomes a bottleneck when we have a possibility to have a SPF on reverse proxy site.

Mixed Server Architecture:



**Data Flow and representations:**



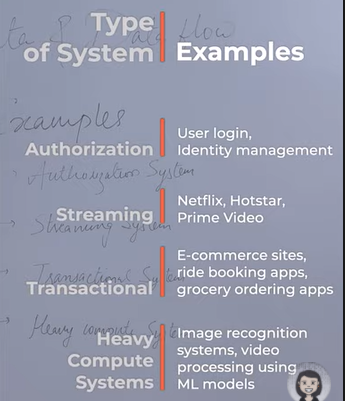
Data Stores: Databases, Indexes, Queues, Cache. (Places where data is needed to be stored):

Dataflow methods: Methods like User event calls, Messages (in real time applications) and APIs etc…



Data generator: User/Internal data (meta-data)~system data/Insights (user data populated by system)

Data Factors Affecting System design: Type of data/Volume/Consumption and retrieval and security.



**Types of Databases:** Different databases depending on query requirements and properties of data like (type and volume) provided different ways and features to store data.

Popular types:

* Relational DB:

The 2 factors which helps us decide if one should have a relational DB in the system or not are ACID properties (-) and Schema (How your data is going to be structured). Also the data relationship is well known.

+Here Database with complex relation can easily be designed.

+Inconsistency has no chance here. Even Database constraints helps in putting strict guidelines on how and what should be needed.

+ACID

**When to Use RDBMS:**

Application involving frequent Transactions like banking or monetary applications.

When you are certain that Table structures are future proof in terms of updation.

-Vertical Scaling is supported but not the horizontal one.

-When Table size grows JOINS operation can be an expensive one and can hamper the system performance.

* Non-Relational DB: Schema of stored data is not fixed.

1. Key-Value Store: Example Redis/Dynamo DB/Memcache. Just like a Hash-Map.

+Fast and Provide quick access as most of the data is in memory.

Is generally used by cache with request response pair as Key value pairs.

On technical side, key0value store is generally used for in-memory data caching to speed up applications by minimizing reads and writes to (slower) disk-based systems.

+High performance and scalability.

It is not ideal for a scenario where there are heavy writes/updates. Not considered suitable for applications requiring frequent updates or for complex queries involving specific data values, or multiple unique keys and relationships between them.

1. Document DB: They are usually used when Schema is not future proof and you want a flexibility of keeping dynamic data (might change over time) then this is the best option.

Collection (Doc…)

+It provides high performance over complex join structure in RDBMS as it stores everything in one entity.

+Support heavy read and write.

-Data inconsistency and dirty data as ACID property is absent.

+Highly scalable (Horizontal)

1. Column DB: Sort of midway between RDBMS and Document DB. There is a fixed schema but do not support ACID properties. They are generally used when you have a application involving heavy writes of data like streaming data/event data. (IOT, commodity tracking). They do not support heavy read operation but special kind of reads. Cassandra is a good example of this type of DB. Column are good supporter of Distributed DBs.

* File DB
* Network DB

**Database Sharding:** DB partitioning (Persistent storage scaling via DB or collection partitioning).

Vertical Partitioning: Store every column in different Shards.

Horizontal Partitioning (Actual Sharding): Store bunch of rows / collection’s Doc in 1 shard.

Physical and Logical Shards: Logical Shard is nothing but portioned data (Say out of 1 billion users table, we have 4 partitions (horizontal) each of 250 million based on some sort of key, these each partitions are called shards). Whereas Physical Shard is the actual machine where the logical shards resides.

Advantage of Sharding: Scale up accordance with the customer traffic. Also Increases system performance via query optimization (If we need to query something based on the key we know in which physical shard do we need to search and hence we do not need to run the query over all the records, but instead a shard). Avoids risk of SPF. Also geographically distributed data can be stored in equivalent geo-based DB server (Tinder).

Algorithmic Sharding (Application logic decides target shard) Dynamic Shard (A follow-up registry decides the target shard).

Drawbacks of Sharding: Data distribution logic is needed to be good enough so that even distribution of data is possible. (Otherwise might result in complications for system performance). Sharding is irreversible.

Key Based

Range Based

Directory Based

Hashing

Consistent Hashing

CAP Theorem