SYSTEM DESIGN

(VID 1)

How to start with distributed systems?

Suppose we want to scale a product, below are things one should look for:

1) **Vertical Scaling** : ⇒ Optimize precision & increase throughput with the same resources
- Optimise process & increase throughput with the same resource.

2) **Preprocessing**: (ex: CRON JOB) ⇒ Reduce load during peak hours by preparing data in advance.
- Executing tasks beforehand during non-peak hours

3) **Backups** : ⇒ Keep backups to avoid data loss and single point of failure
- Creating copies of data to restore in case of failures

4) **Horizontal Scaling**: ⇒ Get more resources to distribute the load.
- Adding more servers to handle increased load.

5) **Microservice Architecture**: ⇒ Enhance flexibility, scalability & maintainability
- Breaking down applications into smaller, independent services.

6) **Distributed System (Partioning)**: ⇒ Improve performance, reliability & fault tolerance.
- Distributing data or processing across multiple servers.

7) **Load Balancer**: ⇒ prevent overloading any single server, ensuring efficient resource use.
- Evenly distributing incoming network traffic across multiple servers.

8) **Decoupling**: ⇒ Increase system flexibility & ease of maintainence
- Reducing dependencies between system components

9) **Logging and Metric Calculation**: ⇒ Monitor, debug & analyze system behaviour
- Recording system events and user actions.

10) **Extensible**: ⇒ Ensure future growth & adaptability.
- Designing systems that can easily be extended with new features.

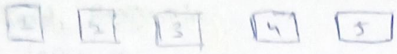
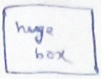
HLD (High Level Design):

Refers to the broad architecture of a system, outlining the overall structure & identifying the main components & their interactions. It serves as a blue print for the system, providing an abstract view.

LLD (Low Level Design):

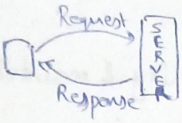
It is a detailed, step-by-step design process that focuses on the implementation of system components.

VID 2

HORIZONTAL SCALING	VERTICAL SCALING
1) 	1) 
2) Load Balancing required	2) N/A
3) Resilient	3) Single point of failure
4) Network calls (RPC = remote procedure calls)	4) Inter process communication
5) Data Inconsistent	5) Consistent
6) Scales well as users increase	6) Hardware limit

VID 3

Scenario: We have a server ⟶ gets overloaded by user requests



So we need to distribute the load.

Load Balancing: distributes requests evenly across multiple servers to avoid bottlenecks.

also ensures we don't have duplicate requests to the same server

- Uses hashing to map request ID's to specific servers, which helps ensure that the load is spread out.

ex: Req. Id ⟶ 10 (r_1) ⟶ hash(r_1) = 13 ⟶ (Let servers = 4)

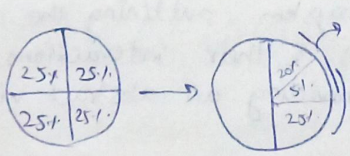
∴ 13 % 4 = goes to 1st server

- ISSUE: Adding / removing servers disrupts the established distribution, forcing a reshuffling of assigned request

ex: 13 % 5 = 3rd server (Server+1)

Now for a particular userId, a particular server is allotted. Each time that user makes request, the server saves it's info. in it's cache to respond faster. Adding / removing server will disrupt the cached data.

ex:



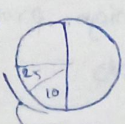
4 servers

↓ (adding 1)

So each server only needs to handle 20% of the load

remove 5% data ⟶ added in next server } ∴ S_1 = 20

30% ⟶ remove 10% data } ∴ S_2 = 20



35% data S_3 has ↓ remove 15% data } ∴ S_3 = 20

S_4 has 40% data, so remove 20% data.

↓

∴ S_4 = 20

&

add new server for remaining 20%

Issue: • 1st server no longer serves 5% of users it used to, so local cache for those users becomes invalid (i.e. useless, so we need to fetch that info. again & re-cache it on a different server that is now responsible for those users.)

• S_2 → like S_1, it lost 10% of users & with it the local cache for those user's info. becomes useless.

• S_4 → lost 20% of it's users. There's a chance that it lost all its previous users & got all other servers' users instead.

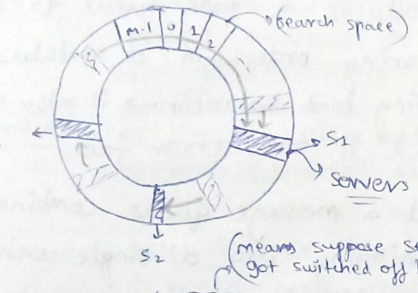
. Like this all servers handle users uniformly but the way we assigned those users are inefficient.

What we need:  → 5% load from each server extracted.

Vid 4

Solution: Consistent Hashing:

Server has Id, request has Id → we hash & map them on a circular "hash ring".



(means suppose server got switched off by mistake)

. Now 'requests' are assigned to servers by finding the next server in clock wise on the ring, ensuring efficient data handling without overloading a single server.

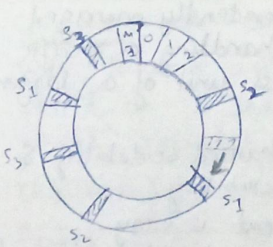
• Now adding / removing a server affects only the nearest requests, reducing the need for extensive data reshuffling. It impacts only a segment of data.

Assuming total servers = N → Theoretically load on each server = $\frac{1}{N}$

but practically, we can have skewed distributions.

(one of the reason = we don't have enough servers)

So to solve this → we can make 'virtual servers'



Multiple hash points (let = k) are introduced to spread load uniformly, avoiding issues with skewed distribution in smaller system.

↳ If we take appropriate 'k' val. → (ex. log N or log m), we can almost entirely remove the chance of skewed load on one of the servers.

This method is widely used in web caches, databases & distributed systems for effective load management & scalability.

Drawback → Increased complexity

Q) Why cant we simply take 5% of load from each server & upload it on a new server?

A) - Manually reallocating data from each server is complex & costly as the system grows.

- Simply moving 5% of data from each server does not ensure uniform load distribution.

Soln.: Consistent hashing automates data reallocation efficiently, supports dynamic scaling, maintains load balance & minimizes data redistribution when servers change.

Vid 5

Message Queue:

- Asynchronous Processing: highlights the benefits of non-blocking order processing where requests are queued, allowing both clients & the system to operate independently without waiting
- Task Queue for load Management: introduces a task queue to handle multiple servers efficiently by storing orders in a database, which allows reassigning tasks (using load balancer → it also ensures duplicate requests are not assigned) if one server fails.
- Integration of Features: concludes that a message queue combines load balancing, persistence and monitoring into a single architecture, essential in systems like RabbitMQ or JMS for effective task distribution (with consistent hashing)

* The Load Balancer has a heartbeat mechanism which monitors server health every few seconds. If a server fails to respond, it releases tasks previously handled by the failed server to others, avoiding duplicated processing. → (property of load balancer)

Vid 6

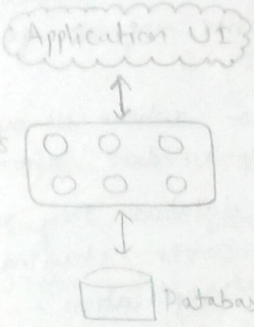
Monolith:

- A single/multiple machine running the computation. It generally combines all functions in a single structure.
- Advantages: → Less Moving Parts
  → Good for small teams
  → Faster in-system calls without network dependencies.
  → Reduced code duplication.
- Drawbacks:
  → Deployment is complex.
  → Complex to manage with increased load. → Tight coupling

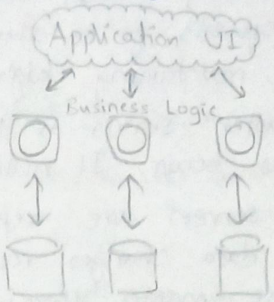
Microservice:

- These are independently managed modules, each handling a specific func. (or) business unit of a larger task.
- Advantages: → Enhanced Scalability & flexibility.
  → Parallel development is easy
  → More straightforward onboarding for new team members.
  → Load-specific scaling for each microservice
- Drawbacks:
  → Tougher to design
  → High coupling btw services should be avoided.

- Any code changes, impacts the entire system, requiring full re-deployment
- It has a risk of full-scale downtime due to interdependencies



- Microservices allow targeted redeployment for individual services, enhancing system stability.
- It minimizes the impact of a single failure by isolating functions.



Vid 7

Database Sharding:

- **Indexing:** It creates a lookup structure (like an index of a book) for specific columns in a database. It optimizes search speed but doesn't alter the physical distribution of data.
- **Sharding:** It involves distributing the data itself across multiple servers (shards) to manage large volumes of data. Each shard manages a portion of data independently, which helps balance the load across servers & makes it possible to handle more concurrent requests.

Horizontal Partitioning: (~ Sharding)

- Data rows are divided across multiple tables/servers based on a specific key, like user ID. Each partition (shard) holds a subset of data rows with the same structure, which allows the database to distribute workload efficiently across multiple servers.
- Useful for large datasets where adding more servers can help balance the load & scale the system.

Vertical Partitioning:

- Columns are split across different tables/servers. For instance, in a user database, general profile information (like names & contact) might be stored on 1 server, while sensitive info (such as payment info) are stored seperately.
- It's beneficial when certain columns are accessed frequently, enabling faster access without loading unnecessary data.

Challenges of Sharding:

- Joins across shards are complex & expensive, as they involve querying multiple servers & merging data, which can lead to performance bottlenecks.

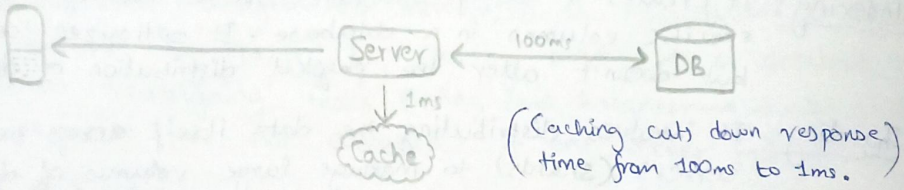
- Sharding's inflexibility can be managed with hierarchical sharding, where large shards are split into smaller sub-shards to balance the load dynamically.

Master-Slave Architecture for Fault Tolerance:

- Describes using master-slave replication for fault tolerance, where read requests go to slave servers, and write requests go to the master, maintaining high availability.
- The 'master' server is the central database where all write operations occur. It maintains the most up-to-date information
- 'Slave' servers are replicas of the master database. They receive & update data changes made on the master server, staying in sync with the master through a process called 'replication'.

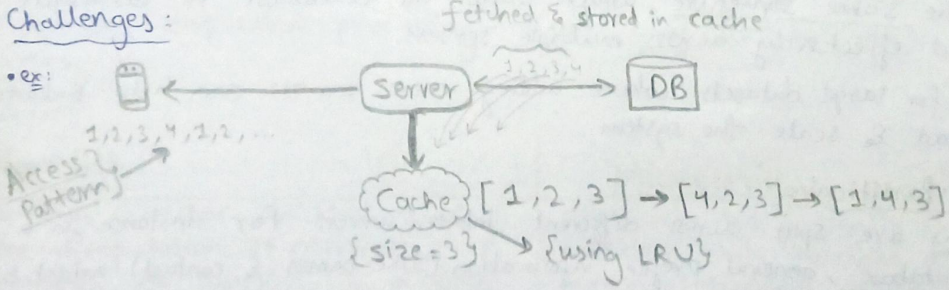
Vid 8

Caching: - stores frequently requested data in a temporary, accessible location, allowing systems to avoid repeat queries to the primary database.



(Caching cuts down response time from 100ms to 1ms.)

- Write Consistency: • Ensuring that both cache & database are updated simultaneously or within a defined delay (like 1min, 1hr).
- Eviction Policy: • Since cache have limited storage capacity, we need strategies like LRU or LFU to remove outdated entries & make space for new data.
- Challenges:



Thrashing: Frequent addition & removal of cache entries which leads to increased latency when the caching policy doesn't align with access patterns.

• Eventual Consistency: Cached data may not reflect the latest database changes/updates, which could be problematic in critical applications like financial transactions.

- Cache Placement:
  - In-memory: local cache present in servers which is running along with my applications
  - Database-integrated: cache for common database queries which is stored in database's server itself.
  - Global-Distributed: → It's an external system which itself is a cache server for handling data across multiple services. ex: Redis
    → This cache is independent & also scalable.

In large scale production, all 3 placement techniques are applied.

- Some Ways to mitigate eventual consistency issues:
  1) Write-Through Cache: Updates are written to both the cache & DB simultaneously, ensuring the cache always contains up-to-date data. This approach can add some latency but guarantees consistency.
  2) Write-Back Cache: Updates are made to cache first, then asynchronously written to the DB. This reduces immediate latency but risks temporary inconsistency if there is a failure before database updates complete.

There are other methods as well, refer them later..

Vid 9

Avoiding Single Point of Failure In Distributed Systems:

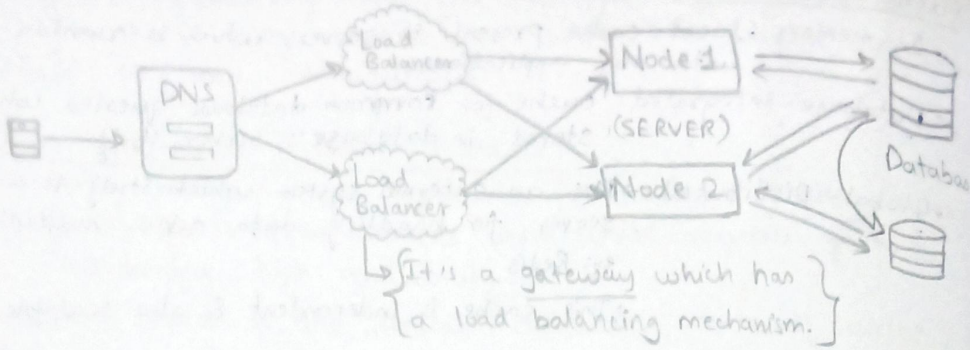
A single point of failure is any part of a system that, if it fails, will stop the entire system from functioning. In distributed systems, such points reduce resilience & reliability.

- Basic Mitigation Strategies:
  → Redundant Nodes: Duplicate critical nodes (ex: database, services) so that if 1 node fails, another can take over
  → Master-Slave Architecture: For databases, the master holds the main data, and slaves replicate it, allowing read requests from slaves & preventing data loss if 1 node fails.

- Load Balancing: Load balancers distribute request among multiple servers. However, load balancers themselves can become SPOFs (Single Point of Failures), so having multiple load balancer, along with DNS to route requests, increases resilience.
- Disaster Recovery: By distributing infrastructure across geographic regions, systems become resilient to localized failures, ensuring continuity in case of regional disasters.

Netflix Chaos Monkey:
Netflix tests it's system resilience by randomly shutting down instances in production, ensuring the system is highly distributed & can withstand node failures without disruption.

ex:



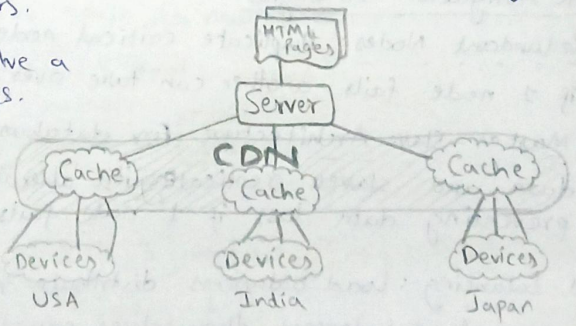
⇒ With backups, the probability of complete failure drops significantly. If a single failure probability is "p", duplicating the node changes this to approx. p^2, making failure events less likely.

⇒ DNS Benefits:
- It enhances resilience in load balancing by providing multiple IP addresses for a single domain name.
- If one load balancer fails, DNS directs new requests to the remaining active load balancers.
- It optimizes connections by routing clients to the nearest (or) fastest responding load balancer based on location.
- It helps distribute requests among multiple servers or regions, preventing overload on any single server/data center.

Vid 10

CDN (Content Delivery Network):

- It's a network of distributed servers designed to deliver content like web pages, images and videos quickly to users based on their geographic location. It enhances performance & reduces latency by caching data closer to users.
- Working: Users initially resolve a website's address through DNS. Instead of a single server, a CDN directs users to the nearest cache server, minimizing the time to access data by reducing the distance travelled.



ex:

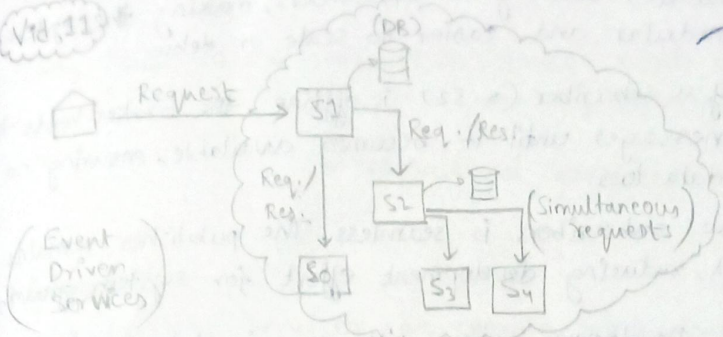
- Benefits:
  - Faster access
  - Meets local regulations on content availability (ex: region-specific streaming rights)

- The best CDN solutions provide 3 things:
  1) Boxes close to users
  2) Follows regulations
  3) Content updated from server

- Ex: Amazon Cloudfront
  - Usually the data sent/stored in CDN is static data like videos, images & files.

- ∴ CDN is a blackbox which stores static content close to all of your clients. It's usually very cheap and very efficient for fast data access.

Vid 11



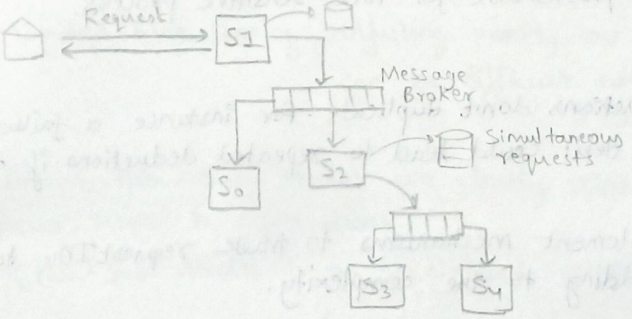
→ Request-Response Architecture:

- Each service performs specific tasks & communicates with others.
- If we use this archi. asynchronously, we have few drawbacks.

- Drawbacks:
  - The order in which responses arrive doesn't matter but can still cause delays or errors if one fails.
  - If a service (ex. S4) fails, the entire chain might time out, causing the client request to fail after a significant delay.
  - This timeout will result in stale data in S1's Database.

(This highlights the inefficiencies of tightly coupled systems.)

Publisher - Subscriber Model:



This model is a design pattern where the publisher (S1) sends messages to a message broker (ex. Kafka, RabbitMQ) rather than directly to other services (S0, S2).

- Key Features:
  - ⇒ Fire & Forge Messaging: Publishers send messages without waiting for acknowledgement, allowing the system to continue working.
  - ⇒ Message Broker Rule:
    - The broker stores & forwards messages to subscribers (S0, S2) even if they're temporary unavailable.
    - Messages are processed when subscribers come online, ensuring eventual consistency.
  - ⇒ Decoupling: Publishers don't need to know the subscribers. Adding new subscribers (ex. S6) is as simple as registering them with the broker.

- Advantages:

⇒ Decoupling Services: Publishers only send messages to the broker, which handles delivery to subscribers, making the system modular and easier to scale or debug.

⇒ Improved Reliability: If a subscriber (ex. S2) is offline, the broker holds the messages until it becomes available, ensuring no data loss.

⇒ Scalability: Adding new subscribers is seamless. The publisher remains unchanged, reducing development effort for system growth.

⇒ Simplified development: Developers work with generic interfaces, reducing complexity in designing service interactions.

- Disadvantages:

⇒ Lack of Atomicity: It's unsuitable for financial systems where consistency is critical. for ex:

- In a ₹950 transaction, if one service deducts the amount but another service fails, data becomes inconsistent.
- A second transaction may cause unexpected results due to partial updates in the system.

⇒ No gurantees on order: messages might not be delivered in the order they were sent, which can be problematic for time-sensitive process.

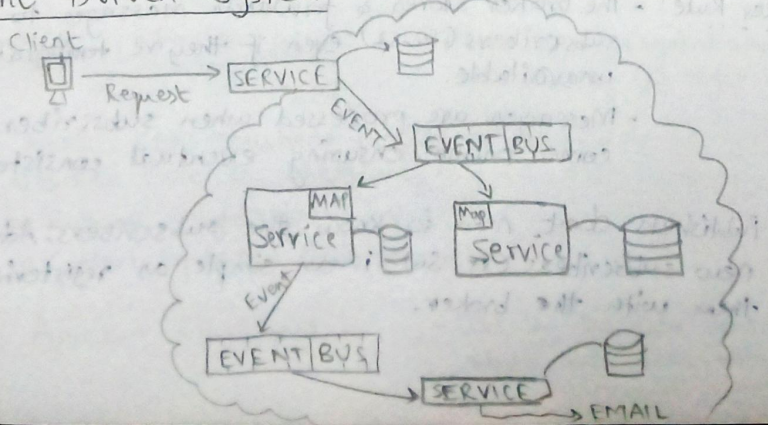
⇒ Idempotency Issues:

- Idempotency ensures actions don't duplicate. for instance a failure in processing a ₹50 debit could lead to repeated deductions if the message is replayed.
- Developers need to implement mechanisms to track requestID's to prevent duplication, adding to the complexity.

- Ex: Twitter uses this model for distributing tweets.

Vid 12

Event Driven System:



- Event-driven architecture (EDA) enables services to communicate via events instead of direct interactions.
- Producers publish events to an event bus, & interested subscribers consume them.
- Key Mechanisms: ⇒ Event Persistence: events are stored in local DBs by services for reliability & debugging
  ⇒ Transactional Guarantees: offers "atleast once" or "at most once" message delivery
- Advantages:
  ⇒ Availability: services can operate independently, even during partial failures.
  ⇒ Replayability: Systems can debug & restore historical states using event logs.
  ⇒ Smooth Service Replacement: Replacing a service is seamless by replaying events to synchronize states.
- Challenges:
  ⇒ Consistency Issues: data btw services might become inconsistent without strict updates.
  ⇒ Complexity: debugging & understanding program flow can be harder due to indirect service interactions.
  ⇒ Limited Fine-Tuning: adjusting priority for handling specific requests can be difficult with an event bus.

- EDA & Pub.-Sub. Model are closely related but they differ in their focus, scope & applications.
  (Pub.-Sub. Model is a specialized version of EDA)

Feature	Event-Driven Architecture	Publisher-Subscriber Model
Focus	state changes & cascading workflows.	Broadcasting messages
Interaction	indirect, multi-layered	direct via broker
Scope	broader architectural paradigm	specific messaging pattern
Use Case	complex, asynchronous systems	notifications or updates
Example	gaming systems, analytics pipelines	News feed updates, system logs.

- When EDA is more than Pub-Sub:

If the architecture involves:

- Event replay (for debugging)
- Complex state transitions (btw services)
- Event-based workflows (spanning multiple service layers)

Vid 13

SQL:

Id	Name	Addr	Age
		23	

Addr ID	City	Country
23		

NoSQL:

ID	Value
123	{ "name": "Ankit", "addr": { "id": 23, "city": ..., }, "age": 23 }

Key Features:

- Flexible Schema
- Built for scalability: supports horizontal partitioning (sharding) to manage large datasets & user load.
- Data Storage: data is stored as 'blobs', grouping related data together for faster insert & reads.
- Optimized for aggregations

Advantages:

- Flexibility in schema
- Horizontal Scalability
- Fast performance due to denormalized data.
- Aggregation support

Disadvantages:

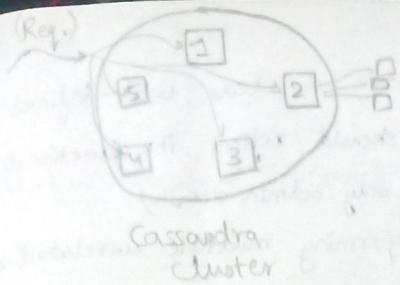
- Consistency Issues: Doesn't gurantee strict ACID properties, unlike SQL.
- Not Read-Optimized: Querying specific columns requires scanning full blobs of data.
- Lack of Relationships: No built-in mechanisms for defining relationships.
- Joins are Manual: Complex joins between datasets are inefficient & must be coded explicitly.

Ex: Cassandra

Quorum: A quorum ensures data consistency in a distributed system by requiring a majority of nodes to agree on a value before it is considered valid.

Sorted String Tables (SSTs):

SST Tables are critical storage mechanism in NoSQL databases, particularly in Cassandra. They're immutable, on-disk structures that store(↓) key-value pairs. (sorted)



SST Working:

- Incoming write operations are stored temporarily in memory.
  - Data is written sequentially for efficiency.
- Memory data is periodically dumped into SST Tables.
  - These tables are immutable, ensuring consistency & enabling fast writes.

- Each record in an SST Table carries a timestamp.
  - When duplicate keys exist, the most recent timestamp determines the correct value.
- Over time, duplicate keys & unnecessary data accumulate across SST tables.
  - The compaction process merges multiple SST tables, retaining only the latest version of each record, optimizing storage & performance.
  - Deleted records are managed using tombstones, which mark records as "deleted" during compaction.

In short, SST Tables enhances performance & storage efficiency in systems with high write & update volumes by ensuring quick writes & batch optimizations.

You may also refer on how "Quorum" is used in Cassandra.

(In short):

- Data is replicated across multiple nodes. (ex. replication factor = 2, means data is stored on two nodes.)
- When a write occurs, the data is sent to all replicas. A quorum is achieved if a majority of replicas successfully record the data.
- When reading data, if the replicas have different versions of the data, the system uses the latest timestamp to return the most recent version.
- If one or more nodes are slow or fail, quorum ensures that operations can still succeed as long as the majority of nodes are functional.
- Hence Quorum ensures eventual consistency, balancing availability & fault tolerance.

Vid 14

API:

- An API (Application Programming Interface) defines a documented way for external systems to interact with your application without revealing internal code.
- It serves as a contract that specifies 'what the API does', 'input parameters', & 'responses', but not how it works internally.

Principles of Good API Design:

- **Clarity in Purpose:** Each API should have a clear, well-defined action
- **Appropriate Naming:** The name of the API should reflect its functionality (ex: 'GetAdmins' fetches only admin details)
- **Minimal Side effects:** APIs should avoid performing multiple unrelated actions.

Best Practices:

- **Parameter & Response Design:** Avoid overloading APIs with unnecessary parameters or responses. Keep it minimum & relevant.
- **Optimize for Performance:** Additional parameters are acceptable if they significantly reduce system overhead.
- **Error Handling:** Clearly define expected errors (ex: group not found, invalid ID) while avoiding overly generic / overly detailed error definitions
- **Atomicity:** Ensure API actions are isolated, complete, & do not rely on partial execution of related actions.

Advanced API Features:

- **Pagination:** divide large responses into smaller chunks using offsets.
- **Fragmentation:** split oversized responses for internal service-to-service communication.
- **Consistency Levels:** Decide whether real-time accuracy (strong consistency) (or) approx. results (eventual consistency) are acceptable based on use case.
  - APIs with heavy loads can implement caching or service degradation (ex: reduced response data) to maintain reliability.

→ Using appropriate HTTP methods (GET, POST) & naming conventions reduces ambiguity & improves routing efficiency.

Common Mistakes:

- Mixing routing & action logic
- Building APIs with side effects (ex: 'setAdmins' creating a group & adding admins simultaneously.)
- Overloading responses with unnecessary data for "future-proofing".

Response of an API:

- **Object as a Response:** → Adv.: - Flexibility for backward compatibility. - Future-proofing - Ease of Maintainence

  → DisAdv.: - Reduced Clarity - Potential for overloading responses.

- List as a Response: → Adv.: - Clear contract definition
  - encourages API versioning
  - promotes minimalism

→ Disadv.: - Frequent contract updates - Consumer migration.

- When to use ?:

→ Use obj. responses when: - The API is designed for long-term use & you expect frequent incremental changes.
- The consumer base is diverse, making b/w compatibility a priority.
- Your API has large no. of dynamic fields

→ Use lists when: - clarity & simplicity are critical, especially for APIs used by external developers.
- the API's functionality is stable, with rare updates.

Vid 18

How Netflix Onboards New Content: Video Processing at Scale

- Challenges in uploading content:

→ Multiple formats & resolutions:
- Videos are processed into various formats (ex: MP4, AVI) & resolutions (ex: 480p, 720p, ...) to accommodate diverse devices.
- This creates multiple combinations for each video, increasing storage & processing complexity.

→ Compression & Quality:
- Using 'codecs', Netflix compresses videos to reduce file sizes while retaining quality.

- Chunk-Based Processing:
- Videos are now divided into 4-second shots. These smaller segments allow → Fine-grained processing across multiple servers.
  ↳ Dynamic grouping into cohesive scenes, ensuring smooth playback transitions.

- Scene-Based Segmentation:

→ Grouping Shots into Scenes:
- Instead of arbitrarily dividing videos, Netflix identifies 'scenes' & groups chunks to reflect the content's natural flow.
- Ex: A car chase scene is processed as one unit, preventing interruptions mid-action.
- Streaming algorithms fetch entire scenes proactively, reducing latency & buffering.

- **Caching with Open Connect:**

Netflix leverages its specialized Content Delivery Network (CDN) called "Open Connect" to store & serve content efficiently:

→ Localized Caching:
- Open Connect boxes are deployed at ISPs worldwide.
- These boxes store regional content, minimizing the need to fetch data from US servers.

→ Reduced Latency & Bandwidth Usage:
- Requests for popular content are served directly from the nearest Open Connect Box, speeding up delivery.
- Over 90% of Netflix traffic is served through these localized caches, improving scalability & user experience.

- **Innovative Features:**

→ Predictive Algorithms: Netflix analyses user behaviour to classify movies into 2 categories

- Sparse Content: ex: documentaries / timepass videos where users skip around randomly.
  ↳ Netflix streams only the requested chunks without preloading future scenes.
- Dense Content: ex: movies (or) series watched linearly.
  ↳ Netflix preloads upcoming scenes proactively to ensure seamless playback.

→ Content Pre-Fetching:
- For dense content, Netflix anticipates what the user will watch next & preloads it.
- Pre-fetching happens during idle moments or at off-peak hours (ex: 4AM), ensuring minimal impact on active users.

Vid 19

Estimation Question: Estimate Youtube's daily video storage requirements

(All are assumptions)

1) No. of users = 1B
2) Ratio of uploaders : users = 1 : 1000 ∴ No. of uploaders = $\frac{1B}{1000}$ = 1M
3) Avg. length of video = 10min
4) Total uploaded video = 10 × 1 = 10M min.

ex: 2 Hr movie avg. size = 4gb
optimised codec & compression savings = 90%
∴ 2 Hr youtube video avg. size = $\frac{4}{10}$ = 0.4GB
∴ 1 min youtube video avg. size = $0.4/120$ GB = 3MB

∴ total video size = 10M × 3MB = 30TB

{More accurate no.'s = more close to original solution}

ESTIMATE CACHE REQUIREMENTS FOR VIDEO METADATA

(Thumbnail & title of the video)

Assume compressed size ≈ 100KB

Now, we want thumbnail of only frequent/popular videos (assume 10% of total videos) → let 80M

∴ 80M × 100 KB = 8TB storage = 8TB RAM

(total uploaded / popular vids)

(we will be using 16GB ram)

∴ 8000GB / 16GB/node = 500 nodes → in my cache system

Now, we want redundancy (as we are serving info. all over the world)

∴ 500 × 3 × (2) → we are using at 50% efficiency, so that it does not get overload other nodes if 1 node gets crashed.

= 3000 nodes

ESTIMATE NO. OF PROCESSORS REQUIRED TO PROCESS VIDEOS

We need to process '10^7 minutes' a day. (every day uploaded)

(1 million videos a day, & 10 min. each) (processor talks in terms of size)

∴ $\frac{10^7}{24} \times 1000$ GB = $\frac{10^7}{24}$ GB in 1 hour → in seconds

$\frac{10^4}{60 \times 60 \times 24}$ GB = $\frac{100 \times 100 \text{ MB}}{6 \times 6 \times 24}$

(min, sec)

≈ 100MB/sec

(Now if we take a computer & we try to process the video through it, we will have 3 steps)

↳ Read + Processing + Write

50ms = (10ms) (20ms) (20ms)

{×10 (= 1000MB/sec)

∴ We need 50ms to process 1MB of data.

(& we have 1000 MB of data/sec.)

∴ Effectively = (1000 × 50 × 10^{-3} sec) of work to do/second. (per)

= (50 seconds of work to do per second)

∴ ⇒ We need 50 processors.

[Since we will write data at many places & we have diff. video formats as well, so keeping a factor.]

(Refer references)

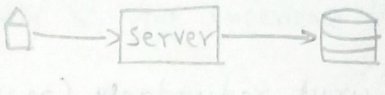
Vid 20

Modern databases must handle massive volumes of writes efficiently without sacrificing read performance.

1) The Need for Write Optimization:

- **Challenges with traditional databases:**
  - Databases like those using B+ trees have efficient reads & writes (O(log n)), but frequent writes generates high IO overhead.
  - Every write requires immediate acknowledgement & sorting, which slows performance.
- **Logs for Faster Writes:**
  - A log behaves like a linked list, allowing fast sequential appends (O(1))
  - Writes are temporarily stored in memory & later flushed to disk in bulk, reducing IO operations.

2) Managing Chunks:



- **What are Chunks?**
  - Chunks are small, sorted groups of data created when logs are flushed to disk.
- **Why use Chunks?**
  - When enough writes accumulate in the log, they're sorted & stored as a chunk.
  - Chunks make writes fast by allowing frequent, independent flushing of data.
  - However, having too many chunks can slow down reads, as each chunk must be searched sequentially/binary wise (depends on context)

3) Compaction: Key to Efficient Reads

- Compaction is the process of merging smaller sorted chunks into larger, more manageable ones.
- ex: 2 chunks → [1,3,5] and [2,4,6] } after merging : [1,2,3,4,5,6]
- **Benefits:**
  - Reduces the no. of chunks, speeding up read operations
  - Consolidates data into larger, sorted structures, making binary searches possible.
- **Trade-offs:**
  - Compaction requires computational overhead & is done as a background process to avoid slowing down active database operations.

4) Bloom Filters For Faster Searches

- A bloom filter is a probabilistic data search that checks if an item might exist in a dataset.
- It can quickly tell → "Definitely doesn't exist" (or)
  → "Might exist" (with a small chance of a false +ve)

- How it works in DBs?
  - Each chunk has a bloom filter that indexes its records.
  - When querying → If the Bloom filter says "doesn't exist", the chunk is skipped.
    → If the Bloom filter says "might exist", the chunk is searched.

- Adv.:
  - Reduces unnecessary searches across chunks
  - Uses minimal memory while significantly speeding up queries.
- Limitations:
  - Bloom Filters can give 'false positives' but never 'false negatives'.

5) Combining Techniques for Efficiency

- Write Efficiency:
  - Logs enable fast sequential writes without immediate sorting.
  - Data is buffered & flushed in batches to reduce IO overhead.
- Read Optimization:
  - Sorted chunks created during compaction enable binary searches, making reads faster.
  - Bloom Filters further optimize reads by skipping irrelevant chunks.
- Compaction Strategy:
  - Smaller chunks are merged into larger ones, balancing write speed & read efficiency.
  - ex: two 6-record chunks merge into one 12-record chunk. (like 2048 game)

6) Key Concepts & Terminology:

- Log-Structured Merge Trees (LSMTs): A database design combining logs for fast writes with sorted structures for efficient reads.
- Sorted String Tables (SSTs): Final, sorted arrays created during compaction, enabling fast search operations.
- Compaction: The process of merging smaller chunks into larger ones to optimize read performance.
- Bloom Filters: Probabilistic tools that improve search efficiency by reducing unnecessary chunk scans.

7) Real-World Applications:

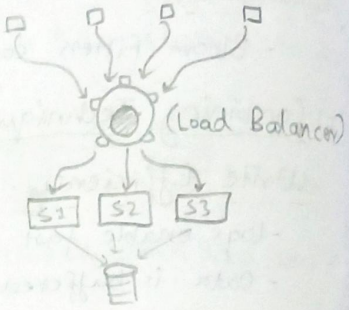
- Write-heavy systems like social media platforms.
- Adv.:
  - Handles billions of records without overwhelming the database.
  - Reduces latency for both writes & reads, even at scale.

Vid 21

Distributed Consensus & Data Replication Strategies on the Server

1) The Prob. of Single Point of Failure:

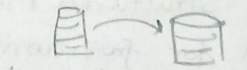
- **Scenario:** → Multiple clients send req. to a load balancer, which forwards them to servers.
  → A single DB is responsible for all these requests.
- **Risk:** → The DB acts as a "single point of failure". If it crashes, the entire system becomes unusable.
- **Solution:** → Introduce "data replication".



2) Synchronous vs Asynchronous Replication:

- **Async. Replication:**
  → Changes are copied to the backup DB after the primary DB processes them.
  → Adv: low load on the primary DB.
  → Disadv: backup data maybe 'out of sync' if the primary DB crashes before replication.
- **Sync. Replication:**
  → Changes are copied to the backup DB simultaneously with the primary databases.
  → Adv.: gurantees consistency btw primary & backup.
  → Disadv.: increased latency, as the system waits for both DB's to complete the operation.

3) Master - Slave Architecture:



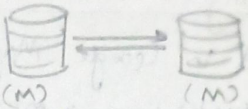
- → Master: Handles all write operations.
  → Slave: maintaines a copy of the master's data & servers read requests.
- **Adv.** → READ SCALING: multiple slaves can handle read requests, reducing the load on the master.
  → FAULT TOLERANCE: if the master fails, a slave can take over
- **Disadv.:** → Writes are only allowed on the master.
  → Slaves may take time to sync with the master.

4) Master - Master Architecture:

- Both databases act as 'masters', accepting read & write operations
- Changes are propagated between the masters.

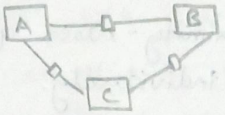
- Adv.: → LOAD BALANCING: Write operations are distributed.
  → REDUNDANCY: Either master can handle operations if the other fails.

- Challenges: → "SPLIT-BRAIN PROBLEM"



  ↳ If communication btw the masters fails, both may assume they are the sole master.
  ↳ Leads to inconsistent states.

  → SOLUTION:- Introduce a 3rd node to break ties & ensure consensus.



- The 3rd node acts as a coordinator, ensuring all nodes agree on the current state.

- If the 3rd node is unavailable, the system prioritizes consistent syncing btw the remaining nodes.

5) Distributed Consensus Protocols:

- We must ensure multiple nodes agree on the same data state in a distributed system.
- Popular Protocols: → Two-Phase Commit (2PC): guarantees consistency but is slow.
  → Three-Phase Commit: Improves on 2PC but adds complexity.
  → Multi-Version Concurrency Control (MVCC): allows to maintain multiple versions of data, allowing older version to be read during updates. (used in DBs like Postgresql)
- Ex: In a food ordering app, funds are 'locked' during a transaction. If the transaction fails, the lock is removed without deducting the funds.

6) Sharding for Scalability:

- Data is divided across multiple nodes, each responsible for a specific range of data
- Adv.: → Reduces the impact of individual node failures.
  → Improves system scalability.
- Each shard can have a slave to take over if the primary node fails.

Vid 22 → (Refer video for clarity)

Designing a Location Database: QuadTrees and Hilbert Curves

⇒ Used in Google Maps, food delivery apps & taxi services.

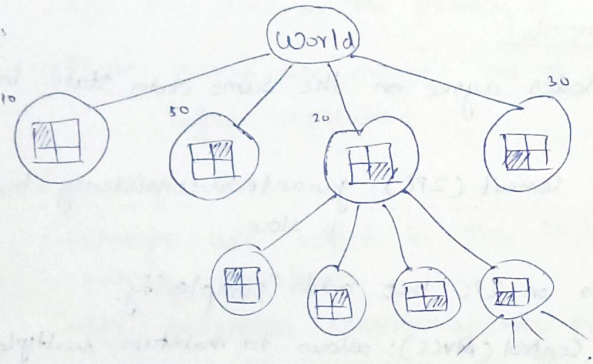
⇒ Key tasks: → measuring the distance btw 2 points.
↳ identifying nearby locations based on a given point

1) Representing Spatial Data:

- **Latitude & Longitude:** → using coordinates is intuitive & scalable.
  → use Euclidean distance $\left(\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}\right)$
- **Challenges:** → Inefficient for proximity-based queries if every point must be checked individually.
  → Requires a more structured approach to represent & query spatial data.

2) QuadTrees for 2D spatial data:

Ex:



QuadTree: - A data str. for dividing a 2D space into hierarchical quadrants.
- Each node represents a region.
- Subdivide into 4 quadrants recursively until a desired granularity is achieved.

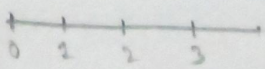
Adv.: - Efficient for breaking down space & narrowing down search regions.
- Reduces the no. of points to evaluate for queries.

Disadv.: - Skewed trees occur in areas with uneven distributions. (ex: dense cities vs sparse rural areas)

3) Converting 2D to 1D: Hilbert Curves

- Range queries are easier in 1D (ex: segment trees). The challenge is to map 2D spatial data onto a 1D representation while preserving proximity.
- Hilbert Curve Solution: • A space filling curve that maps a 2D plane into a 1D line.

(Refer vid.)



• Adv. → enables efficient range queries on the mapped 1D line.
↳ recursive nature allows querying at different granularities.

Vid 23

Data Consistency & Trade-offs in Distributed Systems

(ensures that all copies of data across distributed servers match each other.)

1) Problems with Single Server Archi.

- Single Point of failure
- Scalability issues (hardware limit, etc)
- High latency for global users.

2) Moving to Distributed Systems:

- Disjoint data: servers hold independent data without sharing/syncing.

  Ex: Early Facebook stored data for Harvard & Oxford students on seperate servers without communication.

  Problem: Users couldn't access data across servers, reducing the platform's value.

- Storing copies of data across server resolves latency issues & single points of failure, but the main challenge is: Keeping replicated data consistent.

3) Challenges of consistency in Distributed Systems:

- Network issues, server downtime, or hardware failures can disrupt this process, causing delays (or) inconsistencies.
- 2 Generals Problem: - illustrates the difficulty in achieving absolute certainity in message delivery btw 2 nodes.
  - inconsistencies arise if 1 node commits an update but fails to notify the other, leading to divergent states.

4) Consistency Trade-offs: CAP Theorem & ACID:

- CAP Theorem: A distributed sys. can provide only 2 of the following 3.
  - Consistency (C): all nodes reflect the same data.
  - Availability (A): each request receives a response, even if some nodes are down.
  - Partition Tolerance (P): The system operates despite communication failures.

Trade-off: systems often prioritize availability & partition tolerance over strong consistency.

- ACID Guarantees:
  - Atomicity: transactions are all-or-nothing.
  - Consistency: Ensures database integrity before & after a transaction.
  - Isolation: transactions don't interfere with each other.
  - Durability: Once committed, changes persist.

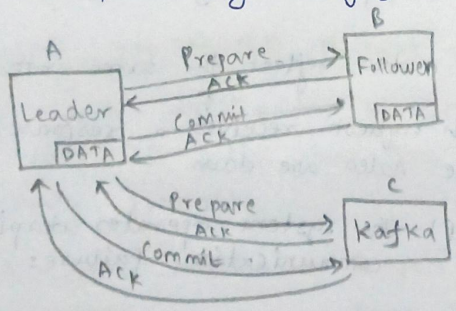
ACID is challenging to achieve in distributed systems without sacrificing availability.

5) Solutions for Consistency:

1) Synchronous vs Asynchronous Updates:
- Sync.: ensures consistency but increases latency.
- Async.: faster but allows temporary inconsistencies.

2) Leader-Follower Architecture:
- A designated leader handles writes, propagating changes to followers.
- Adv.: simplifies consistency management.
- Drawback: Followers may lag behind, creating inconsistencies.

3) Two-Phase Commit (2PC): → (Refer video for better clarity)
- A distributed transaction protocol ensuring consistency across nodes.
- Steps:
  - Prepare Phase: the leader asks followers to prepare for an update.
  - Commit Phase: After receiving acknowledgments, the leader commits the update.
- Drawback: If acknowledgements fail, the transaction rolls back, potentially delaying the system.



6) Eventual Consistency:
- Data updates eventually propagate to all copies, allowing temporary inconsistencies.
- Prioritizes availability & performance over strict consistency.
- Ex: social media platforms like Facebook & Twitter.