Designing Data Intensive Applications

YouTube - https://www.youtube.com/@MsDeepSingh

CHAPTER 1

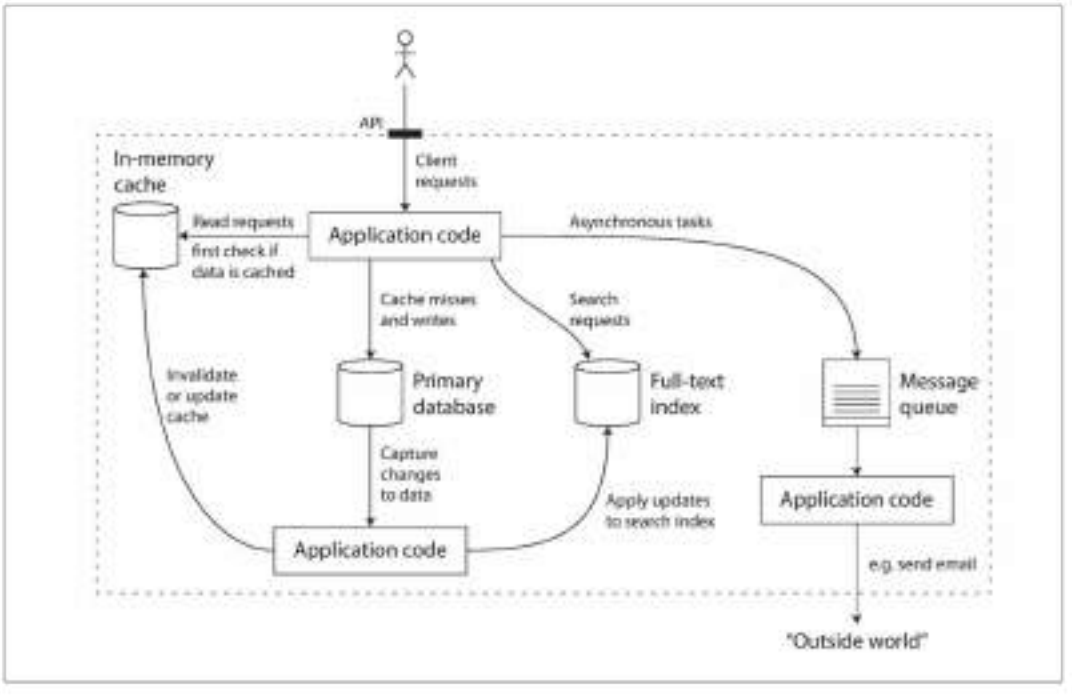
Reliable, Scalable and Maintainable Applications

Reliability → Tolerating Hardware and Software failures
Human Error.

Scalability → Measuring load and performance
Latency percentiles, throughput

Maintainability → Operability, Simplicity and evolvability

One Possible Architecture



Connect on LinkedIn - @msdeep14

How will you ensure below when designing a service →

① Ensure data is correct even if things go wrong?

② Provide good performance to clients even when parts of system are degraded?

③ Handle sudden increase/burst in TPS?

④ How will you design an API for system?

Points to Consider →

① There is no single solution to solve a problem.
It could be possible that ElasticSearch is best search indexing solution available but you went ahead with Solr due to legacy dependencies.

Design can depend on →

① Skills and experience of people.

② legacy system dependencies.

③ time scale for delivery.

④ Organization's tolerance for different kind of risks.

⑤ Regulatory constraints.

Three Important pillars of Software System →

[A] Reliability

System should continue to work correctly even with hardware/software faults or Human errors.

with expected traffic

What can go wrong in system ?

These are called faults.

→ Fault is different from failures.

↳ Watch Error vs Fault vs Failure video on my youtube channel.

↳ Fault means server was unable to serve your request. (probably a specific component of service)

↳ Failure means whole system stopped providing required service to user.

↳ To make systems fault Tolerant, you can design experiments to test system limits.

↳ Watch "Chaos Engineering" video on channel.

Hardware Faults →

any issues in System Hardware – CPU, disk, RAM, router ....etc.

Example → On storage cluster of 10000 disks, avg 1 disk die/day.

To solve this, you should have backups in place

↑ if one component dies, redundant component can take its place.

Software Errors →

Any issue in software which is running on hardware.

Example –

① Bug in software

② You can try to make your system fault tolerant, but if there are issues in third party software our application use, the system can get into failure state.

③ The upstream service is unresponsive.

④ Cascading failures – fault in one component triggers fault in another component

↑ watch "cascading failures" video on YT channel.

Suggestive Resolutions

① Try to close on all unknowns/edge cases during System design.

② Testing

③ Process Isolation
↳ example tabs in Chrome browser.

④ Allow process to crash and restart

⑤ Monitoring + chaos engineering + Alerting + Logging

Human Error → Humans are unreliable ☺

① Experiments in production should be well monitored.

② Right amount of abstraction.

③ Thorough Testing.

④ Monitoring + Alerting. + Best training practices.

So what do you think?

Is Reliability IMPORTant?

Scalability → what happens if your system's load is increased by 10x, will it be reliable?

> It is used to describe system's capability to handle increased load on system.

What is load? → TPS for web server, read write ratio on database, hit rate on cache.

Example Twitter → How a tweet published by user reach his/her followers?

↳ followers asks for new tweet?

↳ user pushes tweet to all its followers?

⇒ Watch similar analogy in "Instagram NewsFeed" video on YouTube.

Here in case of twitter, distribution of followers per user is key load parameter to think around scalability.

Performance → Is system working as per our expectations?

→ What is TPS that can be handled by server for given CPU and memory?

→ What is latency to serve API response?

Response Time determination → p50, p90, p99, etc.

lets say response time = 1 second, then p90 means 90 out of 100 requests take <1sec and other 10 requests take 1sec or more.

How to handle LOAD? load is increased but performance is not impacted.

- → vertical scaling
- → Horizontal scaling

Automatic Scaling without human interventions → Auto Scaling

→ There is no single solution that can be developed for all large scale systems, it varies use-case by usecase.

Maintainability

maintain existing software - fixing bugs etc {oncall}

① Operability → making life easy for operations.

- ↳ Monitoring health of systems
- ↳ Debugging system issues
- ↳ software updates
- ↳ System Automation
- ↳ Oncall Rubooks and System dashboards .....

② Simplicity → Managing complexity.

- ↳ How different components are coupled?
- ↳ hacks introduced in system as short term solve.
- ↳ Complex systems makes testing a cumbersome task.
  - ↳ possibility of introducing a bug.

How can you achieve it?

↳ Abstraction

example → Java abstracts out memory clean up logic.

③ Evolvability → making changes easy.

System requirements change and it will requires modification in existing system, how easy is this process?

Subscribe YouTube Channel for more Such content.

Channel - MsDeep Singh

Happy Learning

Designing Data Intensive Applications

CHAPTER 2

Data models and Query Languages

Relational Model - Data organized in relations ~ tables in SQL.

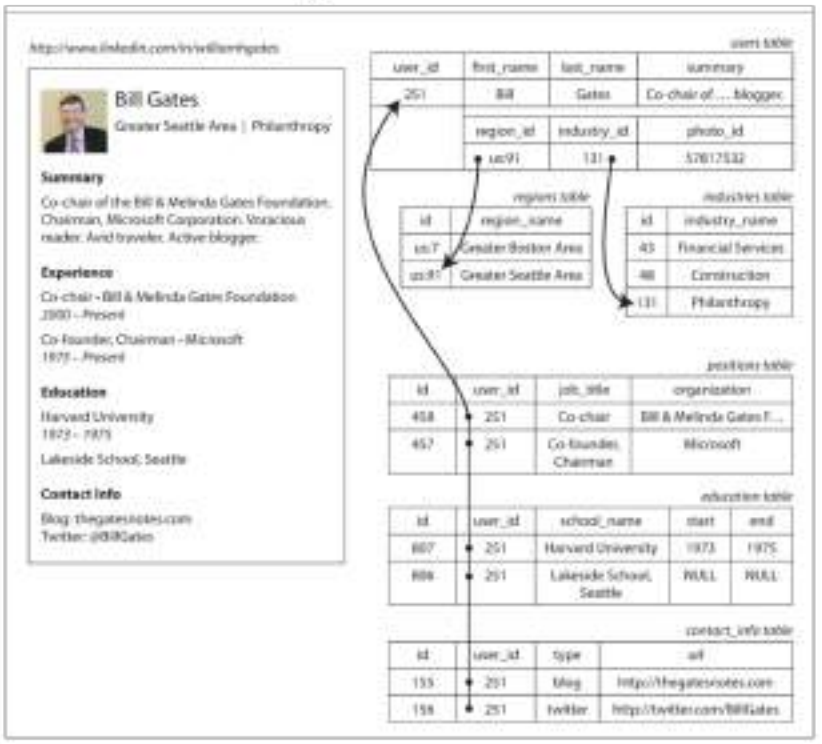
Why NOSQL?

① More scalability as compared to SQL databases

② Open source instead of paid software

③ Query operations that are not supported by SQL.

The way we write code → most applications use OOP paradigm to write code and you'll require extra translation layer between objects in code and data models in SQL tables.



→

```
{userId : 251,
 first_name : Bill
 last_name : Gates
 positions :[
{jobtitle : Co-Chair }
{jobtitle : Cofounder }
]
...
}
```

Image - Wikimedia Commons

Fetch profile for Bill Gates

① Relational → perform multiple queries in tables or perform complex join

② Document → all information at single place.

Comparison of document Model vs Relational Model

① Application code simplicity →

↳ It will depend on application, what is the usecase it is solving.

↳

Property	Document	Relational
① Data in application ?	→ requirement is document like structure. → poor joins support.	requirement is many to many relationships.
② Schema flexibility	→ Don't enforce strict schema. → schema on read - schema is maintained in application code. → Adding new fields is easy task.	→ strict schema → schema on write. → Adding new colums requires proper migration strategy ↳ could be a real pain point if data size is huge. → If all records are expected to have same structure, RDBMS is good way to enforce it.

③ Data locality for queries	→ document is stored as continuous string. → If app require entire doc to be read, it is useful.	→ If data is split into multiple tables and entire data needs to be retrieved, there are performance issues. → Oracle has feature "multi-table index cluster tables" which provide locality property in RDBMS.

Similarity

↳ There can always be feature similarity between SQL & NOSQL. It totally depends on usecase we're try to solve.

↳ NOSQL database has support for joins.

↳ SQL databases can store data in document form.

Query Languages for Data

SQL - declarative query language

Declarative Language	Imperative Language
→ You specify pattern of data you need. ↳ but you don't specify how these results are achieved. ↳ Database system's query optimizer to decide how to gather this data.	→ Tells computer to perform certain operations in certain order.

r ect

collect

Map Reduce Querying

→ programming model for processing large amounts of data.

↳ in later chapters → CH 10

Graph Data Model

↳ Document is good if there are mostly 1-many relationships or no relationships between records.

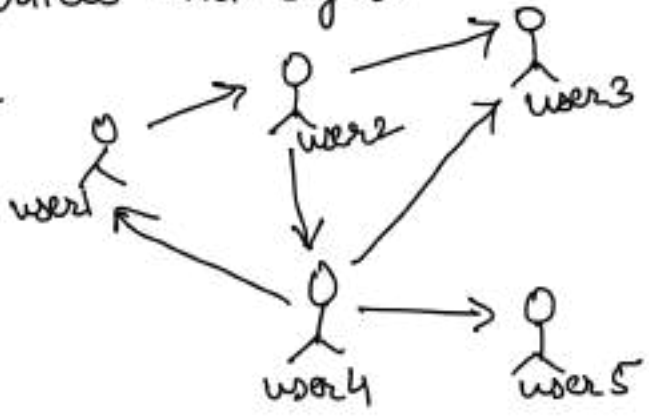
↳ many-many relationships can be very well handled by RDBMS.

↑ But what if these relationships become very complex and nested.

∑ Then graph data model is more suitable.

Graph → vertices and edges.

Example



↳ Graph provides flexibility in data modelling as compared to RDBMS.

Example Neo4j

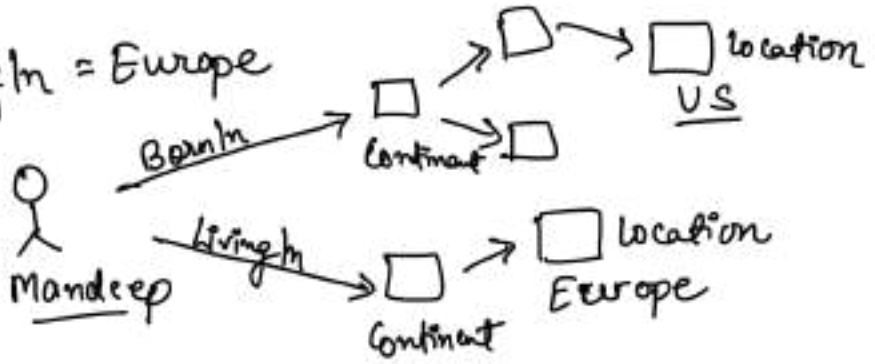
query language – cypher (declarative)

example- Find all people who immigrated from US to Europe

↳ BornIn = US

&&

LivingIn = Europe



Graph Databases compared to Network Model

Network	Graph
① schema specifies record type could be nested within each record type.	① No such restriction. A vertex can have edge to any other vertex.
② To reach particular record, traverse the access path.	② You can directly go to any vertex by its unique identifier.
③ Children of record are ordered set.	③ vertices & edges are not ordered.
④ Imperative queries	④ Support for both declarative & Imperative queries.

Stay Tuned for follow up chapters.

Subcribe for more. Youtube – MsDeep Singh

Happy Learning 😊

Designing Data Intensive Applications

CHAPTER - 3

Storage And Retrieval

How will data be stored in database?

How data will be retrieved from database?

Data Structures that power your Database

Why do I care what's happening in backend? ⇒ It helps in selecting a data storage for your system. There are lot of options these days ☺

Log file → Assume you're keeping a key value pair and store in text file in append mode.

For any new write query, you'll just append an entry.

How will read value for given key?

↳ Scan entire file for occurence of key? → that will be O(n) operation

↳ It's not efficient if data grows to large scale

↳ You'll need indexes (watch youtube video on Indexes on channel)

create indexes for data written.

↳ Should I create indexes for all the data? It will make read faster

→ NO, it will make write slower. Along with append operation, you also need to write data to indexes.

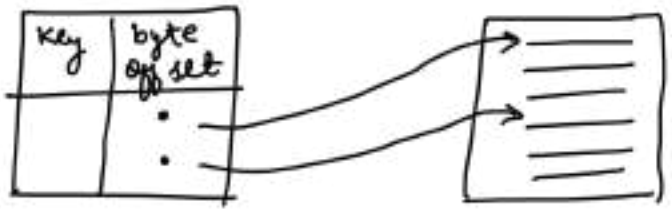
Hash Indexes → How will you index data on disk?

Assumption - Data storage provides only append operation on file

Indexing strategy →

↳ Create in-memory hashMap.

↳ Every key is mapped to byte offset in file.



↳ It is used by Bit cask (Storage engine in Riak)

↳ In memory Hash map helps if there are limited no of keys.

↳ Example - maintain a counter how many times video is played.

↳ How will you ensure you're not running out of disk space (since you're only appending)

↳ ① break the log into segments of certain size.

↳ ② perform compaction - remove duplicate keys in log and only keep recent update.

V1:23	V2:25	V1:31
V3:73	V1:99	V2:50

→

V1:99	V2:50	V3:73

↳ Each segment has its own in-memory hash Table.

↳ Find value for key: check most recent segment's hash map.

↳ If not found, check 2nd most recent.

Implementation Details

① File format – Binary → encodes the length of string in bytes, followed by raw string.

② Record Deletion – append special deletion record to data file.
↳ called Tombstone {watch Discord Msg storage video}

③ Crash Recovery – Maps in memory will be lost in events of crash.
↳ keep snapshot of segment's hash map on disk.

④ Partially written records – use checksums, helps in identifying corrupted parts and delete them.

⑤ Concurrency – Data files are append only → immutable.
↳ can be read concurrently.

Why not update the key instead of appending?

① Much faster than random writes, especially on magnetic spinning disks.
② concurrency and crash recovery is simple.

Limitations

① Hash Table must fit in memory.
② Range queries are not efficient.

SSTables and LSM-Trees

optimizing the limitations mentioned above ↑
↳ change the format of how segment files are stored
↳ make sequence of key-value pair "sorted by key"
↑
This is called Sorted String Table (SSTable)

↳ Each key should appear only once in merged segment.
(handled in compaction process)

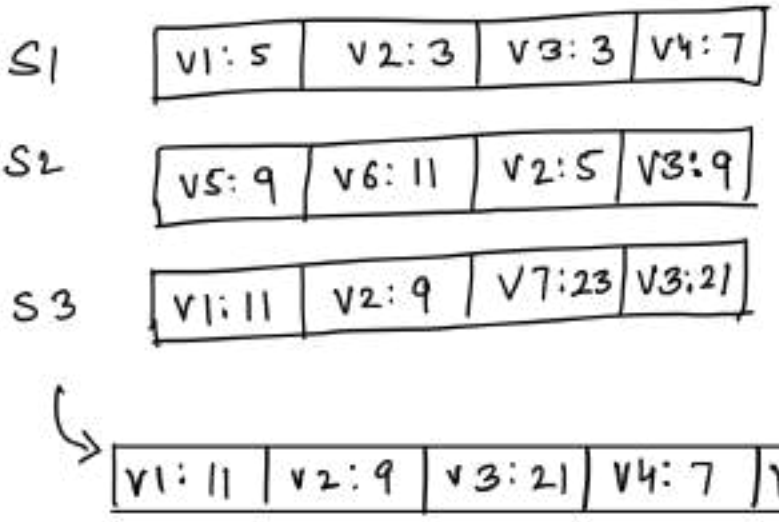
(A) Merging of segments → algorithm similar to merge sort.

① Read input file side by side.

② look for first key in each file.

③ Copy lowest key to output file Repeat again.

④ If same key in multiple segments → pick value from most recent segment.



Search for a key

① Inmemory index is not required for maintaining offset for key.

② maintain offset for some of keys (sparse index) and they traverse between these offsets to find particular key.

③ Group the records for keys you're maintaining in Inmemory index. ↳ saves disk space and reduce I/O.

Construction & maintenance of SSTables

For any write query –

① Add it to in-memory balanced tree structure. – called memtable.

② Write memtable to disk after certain threshold (say 1MB) – called SSTable

③ For read query, search in memtable, then most recent segment & so on.

④ Run merge & compaction time to time.

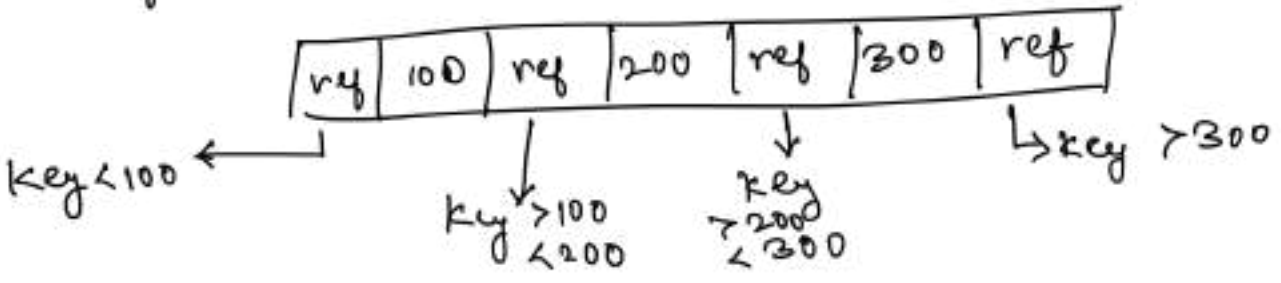
Performance of Database built on LSM Tree

① Reads can be slow as you traverse from one segment to another.
↳ Bloom Filters are used → data structure helpful in approximating contents of set.
Example → will tell if key is present in dB or not saving traversing DB.

② Compaction & merging → how to determine order & Timing.

Size-Tiered	leveled compaction
HBase. Cassandra uses both.	LevelDB, RocksDB
smaller SSTables are successively merged into larger SSTables.	key range is split up into small SSTable and older data moved to separate levels.

BTree → It is not "Binary" Tree.
↳ key value pairs sorted by key. → on disk
↳ BTree break database into fixed size blocks.
↳ Each Block can be identified by address, using which one page refer to another.



Handling Crashes in BTree → what if hardware crashed at time of any update?
It maintains additional data structure → WAL (Write ahead Log)
↓
append only file - every modification to BTree is written here before actual update.

Write amplification → one write to DB resulting in multiple writes on disk over course of DB's lifetime.

LSM Tree	BTree
① Typically faster for writes ② It can sustain higher write throughput due to less write amplification ③ Can be compressed better ④ Compaction process can affect ongoing read/write. ↳ more observable at high throughput.	① Typically faster for reads. ② More write overhead, update entire page. ③ leaves some space unused due to fragmentation. ④ Each key is present a single place. ↳ offer strong transactional support

Full Text Search and Fuzzy Indexes

↳ Lucene example ElasticSearch

ex. to search for mis-spelled words or synonyms.

In-Memory Databases

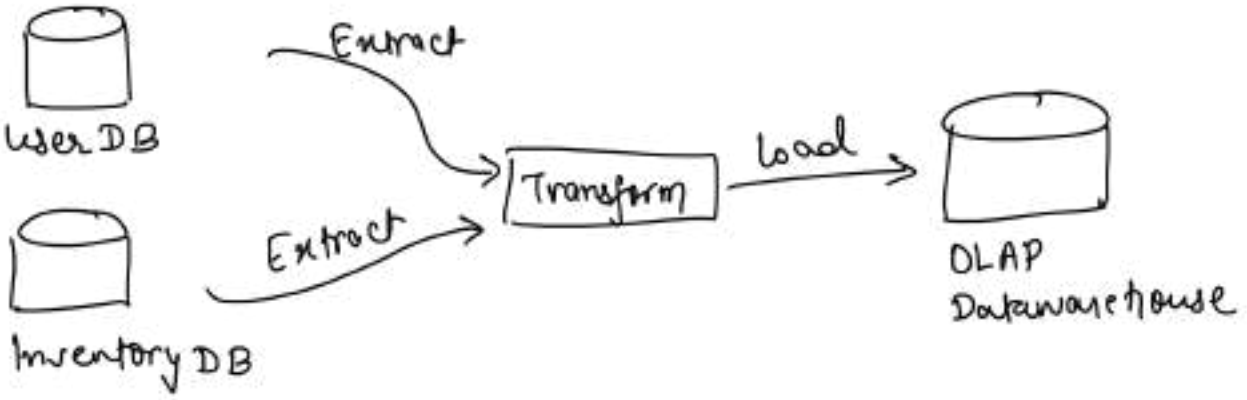
memcached → Indended for caching purpose, it is ok to loose data if machine is restarted.

↳ In memory databases for durability → dump periodic snapshots to disk.

↳ Redis and couchbase provide weak durability by writing to disk asynchronously

Your use-case → Transaction Processing or Analytics ?

OLTP	OLAP
① Read - small number of records per query fetched by key ② write - Random access, low latency writes ③ GB to TB	① Aggregate over large set of data. ② ETL ③ TB to PB



OLTP databases

Column Oriented Data Storage

1. The table can be >100 columns wide but we might be accessing only 4 to 5 columns at a time.
2. To get the data, you need to load all rows matching criteria to these columns
3. Column Oriented → Store values from each column together instead of rows.
4. Column Compression is easier → there are chances that values in column are same for different rows.
5. Example – Cassandra, HBase
   - ↳ not strongly column oriented.
     - → they store all columns for row together along with rowkey.
     - → don't use column compression.

Subscribe for more such content.

YouTube Channel- MsDeep Singh

Happy Learning ☺

Designing Data Intensive Applications

CHAPTER 4

Encoding And Evolution

As application grows over time, new changes will be introduced in application.

↳ How will you rollout these changes without breaking the application and continue serving customers?

↳ Rolling upgrade or staged rollout.

↳ don't deploy at once. Deploy on one node then validate, Deploy on other nodes and validate and so on.

① Backward compatible → new code can work with older data.

② Forward compatible → Old code can work with newer data.

How programs work with data?

① In memory → data to be accessed by CPU on machine via objects lists, hashMaps, etc.

② Send data over network → encode it to some different format example JSON.

Translation from ① → ② is called encoding/serialization/marshalling reverse is called Decoding.

How to encode?

① Language specific support
Java - java.io.Serializable
Python - Pickle
They should not be avoided for any extensive purposes
↳ Encoded in Java will cause problems to decode in Python.
↳ Lack of forward/backward compatibility.

② JSON, XML and Binary variants
↳ support across programming languages.

JSON/XML/CSV
↳ Can't distinguish between number and string.
↳ No support for binary strings → workaround using Base64.
↳ CSV don't have any schema, application define meaning of each row/column.

Binary Encoding
↳ very helpful as data size grows.
↳ example for JSON - BSON, BJSON, etc.

Thrift and Protocol Buffers
↳ Both require schema for data to be encoded.

```
example Thrift
    struct Person {
        1: required string username,
        2: optional i64    favouriteNo
    }
```

↳ req/optional is runtime check and dont reflect in encoding.

→ Thrift has two binary encoding formats
↳ Binary Protocol
↳ Compact Protocol. → takes less space.

Schema Evolution
↳ schema evolves over time. How will you ensure encoding/decoding is backward/forward compatible.

↳ Every field in encoded data is identified by tag numbers.
  ↳ If field is not set, it is skipped from record.
↳ Field names can be changed, but field tags should not be changed as they are used for encoding data.
↳ Each new field added should be given a new tag number.

Backward compatible

↳ ignore the field (datatype helps in skipping byte offset) which are newly added on read.

Forward Compatible

↳ As we ensure unique tag number for each field, it will not cause any issues.
↳ It should not be made required to avoid runtime errors.

Data Type evolution in schema

↳ value can loose precision or get truncated.

Avro → binary encoding format

↳ use schema to specify structure for data to be encoded.
↳ no tag numbers.

```
record Person {
    string username;
    array<string> interests;
}
```

↳ most compact as compared to others.
↳ Avro defines schema in terms of writer's schema (who writes) and reader's schema (who reads)
↳ It works fine as far as they are compatible.

↳ to add/remove value, field should have default value.

↳ maintain version number as schema evolves.

Dynamically generated schema in Avro

↳ You don't have to worry about tag numbers unlike Thrift.

↳ the schema can be generated from your object schema.

Dataflow

① Dataflow through Databases

↳ database will change over time eg. add new column to RDBMS

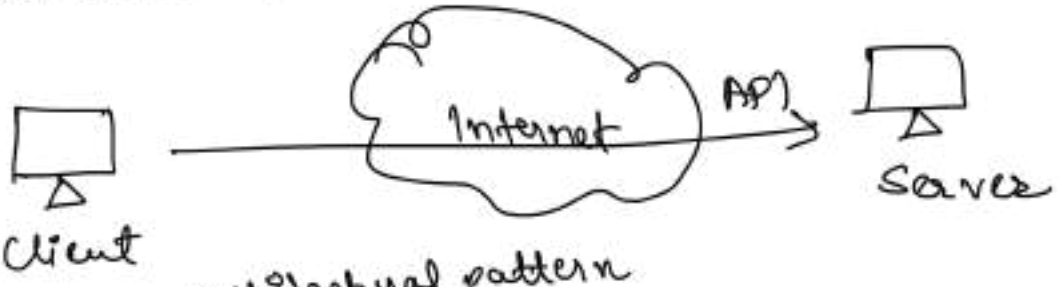
↳ You can specify a default value for newly introduced columns.

↳ Handle specifically in application code as per requirement.

↳ It is good choice as compared to migrating entire db to new schema.

② Dataflow through Services – REST & RPC

↳ communication over network.



REST → Design philosophy that builds upon principles of HTTP (architectural pattern)

SOAP → XML based protocol for network API calls.

eg. GET, PUT

RPC → Remote Procedure call

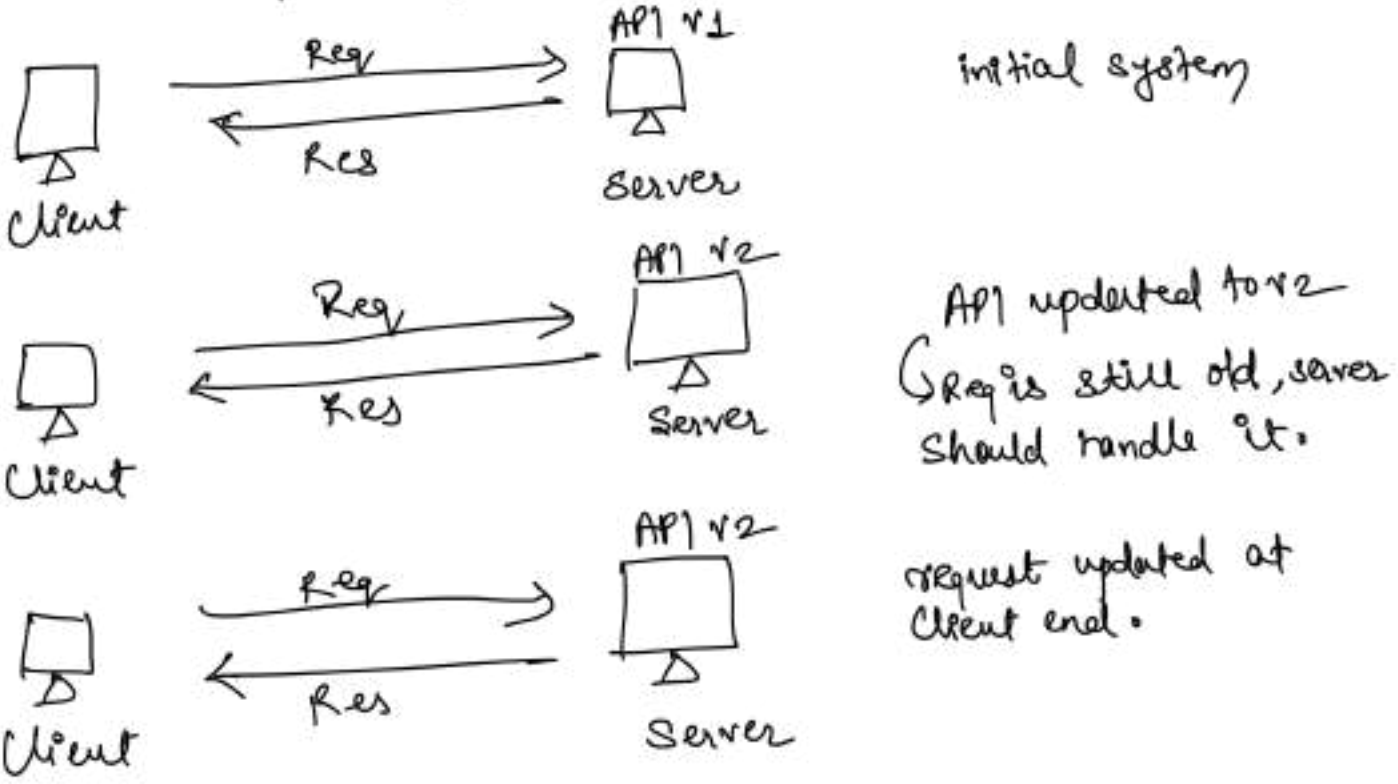
↳ make request to remote network service.

One example- gRPC → support streams, a call can consist of series of request and response over time.

Data evolution → Servers will be updated first and then clients

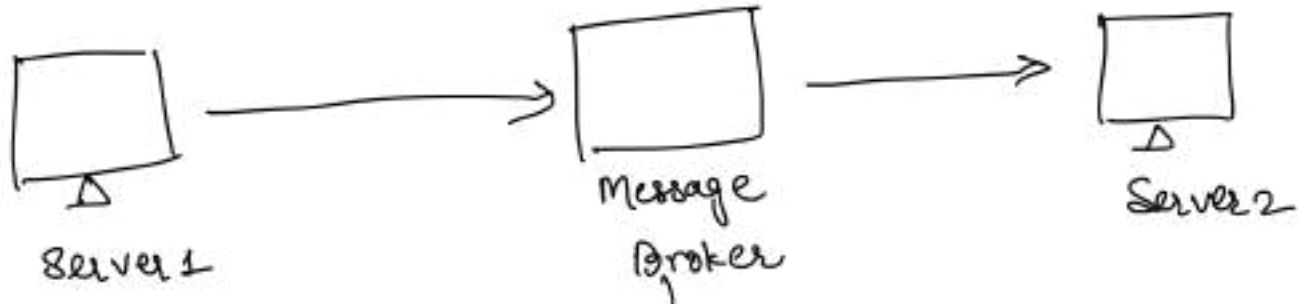
↳ backward compatibility for requests.

↳ forward compatibility for respone.



Message Passing Dataflow

↳ asyn processing



↳ temporary storage, can store msg for sometime if receiver is not available → makes system reliable

↳ Receiver Server can consume msgs on its rate.

↳ One msg can be sent to multiple receivers.
↳ Sender doesn't wait for msg to be delivered to receiver

Example Kafka, RabbitMQ, Amazon SNS-SQS

Actor framework

↳ programming model for concurrency in single process.
↳ Each actor represents one client.
↳ Actors communicate with each other by sending msgs async.

Example
Akka. Erlang OTP

Subscribe for more such content.
YT channel- MsDeep Singh

Happy Learning 😊

Designing Data Intensive Applications

CHAPTER 5

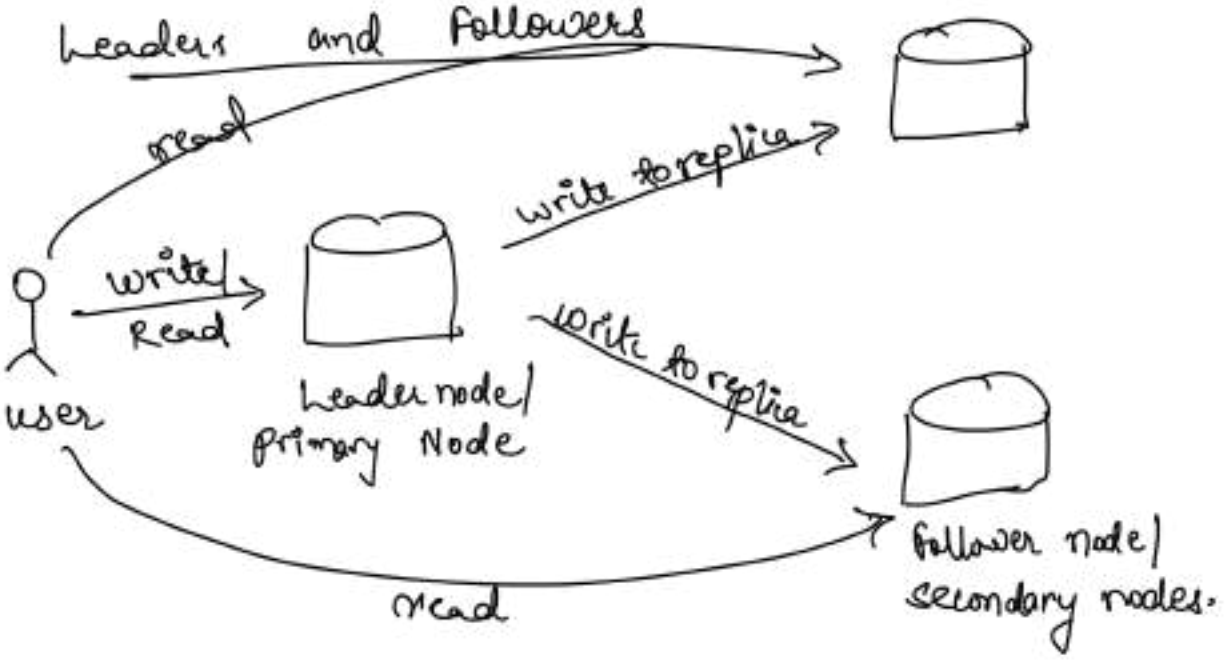
Replication

Keeping copy of data at multiple places.

why?

↳ keep data in same region as users → reduce latency.
↳ Availability → system is operational if certain machine goes down
↳ Can serve more Read TPS.

How to Replicate?



Synchronous vs Asynchronous Replication

① User is sent ack once write is completed on leader & followers	① User is sent ack once write is complete on leader node. Follower nodes are updated in background.
② Data is always consistent	② eventually consistent.

③ write can't be processed if any nodes goes down.

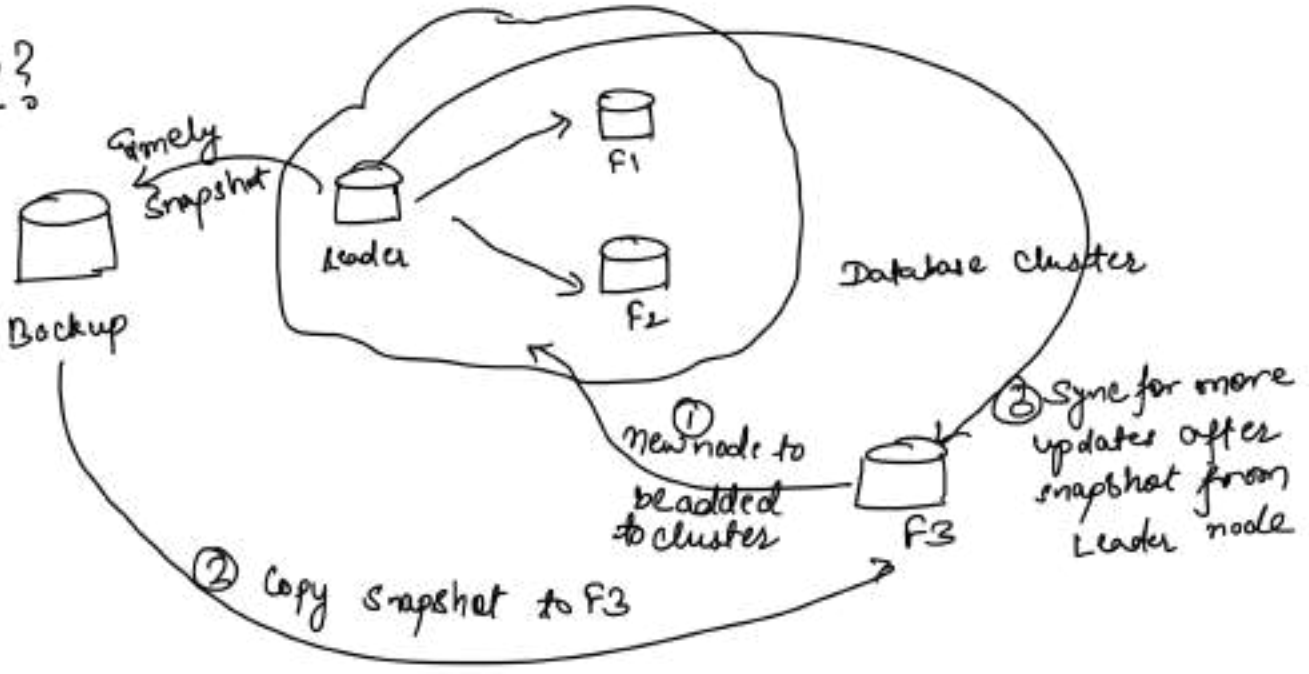
④ for write → leader node should be available.

Adding new follower

↳ in case more replicas are required to handle load.

↳ replace failed nodes in cluster.

How?



Handling Node Failures

① Follower failure

↳ keep log of data changes on local disk. Once restarted, recover using log.

↳ sync with leader node for updates during downtime.

② Leader failure

↳ A follower is promoted as new leader.

ⓐ Identify Leader has failed → health checks

ⓑ Choose new leader → best candidate could be node which has most recent replicated data.

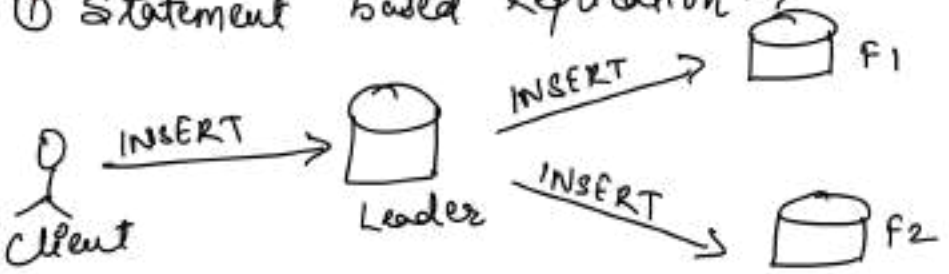
ⓒ System use new leader → new leader accepts writes from clients

↳ If crashed node comes back, it is promoted to follower.

{more Details on above in coming discussions}

Replication logs

① Statement based Replication →



→ Statements received from clients are logged and forwarded to Clients.

Problems
↳ Functions like NOW() or RAND() can store different value on nodes.
↳ If query depend on existing data, they must be executed in same order on each replica. → else can cause different effect.

② WAL – Write ahead log
append only file to maintain the operations on DB → discussed in Ch-3

③ Logical (Row based) Replication

Use different log format for replication & storage engine.
↳ issue in #2 where WAL is used by replication & internal BTree.

① Row Insert → new values for all columns
② Row delete → primary key of row
③ Row Update → new values for columns that changed.

④ Trigger Based Replication
write custom code that is triggered for any operation in DB.

Replication Lag

↳ application can see outdated data in case of async replication.
↳ eventual consistency.

① Read After Write Consistency → "read your own writes"
The user who made an update, will see recent data on read.
This might not be the case for other users.
↳ @ Decide when to read from leader and when from Followers.

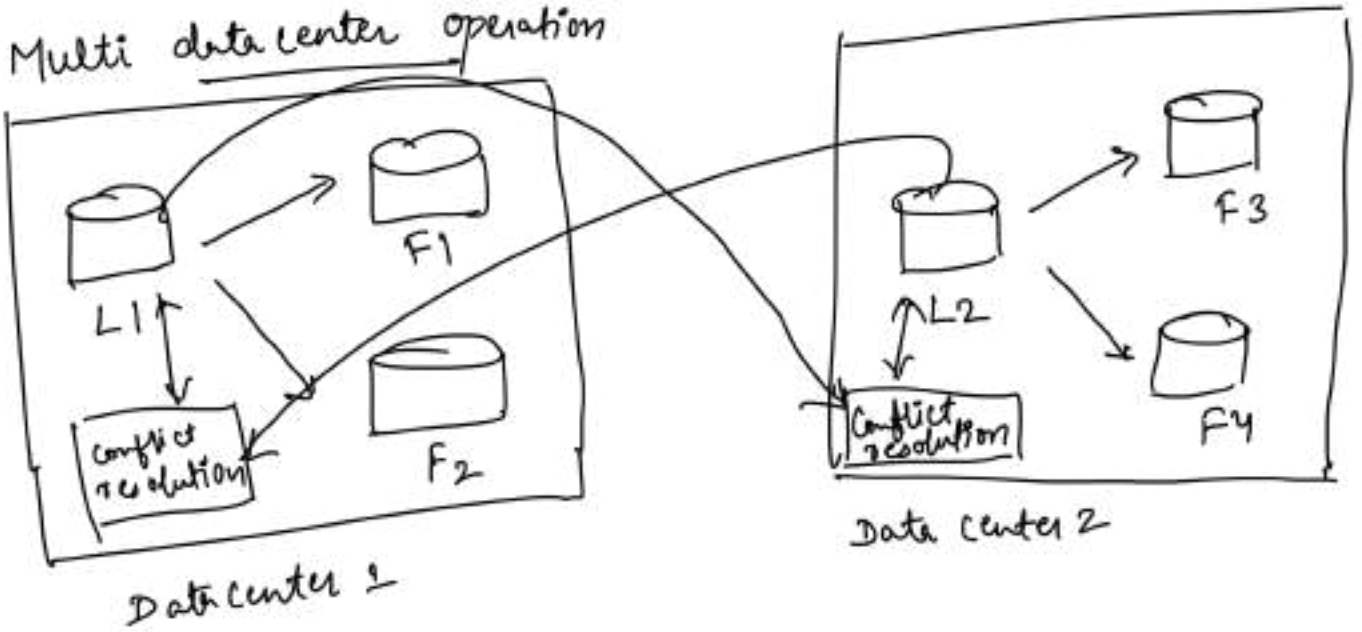
⑥ For 1minute after write, read only from leader.
what if same user reads info on multiple devices?
↳ can user requests routed to same datacenter to handle this.

② Monotonic Reads
Users gets different data on subsequent reads.
↳ can same user always read from same replica.

③ Consistent Prefix Reads
If writes are performed in certain order, reads should happen in same order. → violation of casuality.
→ generally happens in partitioned databases.

Multi Leader Replication
↳ write can happen on any of leader nodes.
↳ Each Leader simultaneously acts as follower to other leaders.

① Multi data center operation



Benefits
① latency improvement - write can happen in specific data center.
② Fault Tolerant → data center outage.
↳ network issues

Downsides
① Same write can happen in both data centers concurrently.
↳ requires conflict resolution.

Examples

↳ Clients with offline operation – calendar application on multi-devices.

↳ collaborative editing – Google docs – conflict resolution for concurrent users.

Handling of write conflicts

① Conflict Avoidance → avoid conflicts to occur.

↳ request from particular user are always routed to same leader
  ↳ single reader in user's view

② Converging to Consistent State

↳ assign each write a unique ID. example Timestamp.
conflicts can be resolved basis "Last Write Wins".

↳ assign each replica unique ID.
writes from higher number replica takes precedence over other.

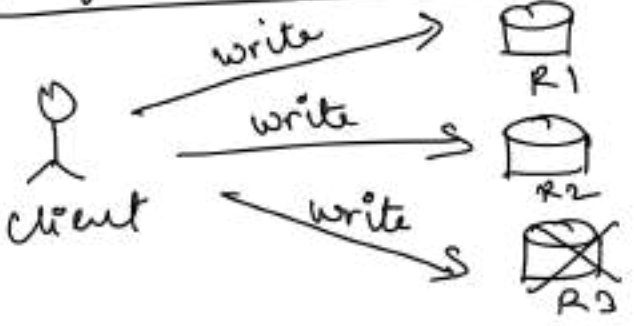
③ Custom Application Code

↳ write your custom conflict handler.

Leaderless Replication

↳ any replica can accept writes from clients.
ex – cassandra, Voldemort

What if a node is down on write?



↳ only R1 R2 are available
↳ write is successful basis quorum
↳ on Read as well, request is sent to all nodes, and then recent value can be picked up basis version no.

How will be R3 in sync once it comes online?

① Read Repair → on read, we identified which Replica has stale data, so write back to R3.

② Anti-entropy process → background process to check differences between replicas.

Quorums

n - replicas
w - no of nodes to confirm on write for it to be success
r - no of nodes to be queried for read.

$w + r > n$

↳ we'll get up-to date reads.

General recommendation

n → odd number
w, r → $(n+1)/2$

① If you set lower values of r and w (r or $w <= n$) → possibility of stale reads.

② even with ($w + r > n$), there is possibility of stale reads.

ⓐ Sloppy quorum →
↳ In case of network disruptions
↳ will you return errors?
↳ OR return response (maybe stale data) without reaching quorum.
↳ Sloppy quorum

reads and writes still require r and w successful responses but the response might not be from designated node.
↳ return temporary respone and once node is back, restore it to correct state.
↑ hinted handoff.

ⓑ Two concurrent writes →
Multiple nodes will have confusion around which is most recent write.

ⓐ Last write wins → each replica stores most recent value and overwrite older values.

↳ as we say writes are concurrent, so write order is undefined. Attach timestamps to each write and choose most recent one.

↳ This might lead to data lose.

↳ Approach for handling this could be to assign version numbers to each write operation and subsequently merge basis that on re-writes.

↳ The old values are deleted directly, kept in database with help of delete markers - called tombstones.

Hope you enjoyed the chapter Summary.

Subscribe for more such content.

YT Channel - MsDeep Singh

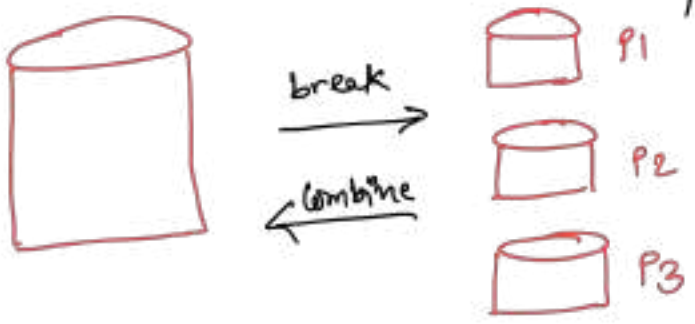
Happy Learning ☺

Designing Data Intensive Applications

CHAPTER - 6

Partitioning

↑ break data into parts ~~partitions~~



why? to scale the system. After certain point, data can't fit into single machine.

↳ the partitioned data should also be replicated on multiple nodes for fault tolerance. → high availability

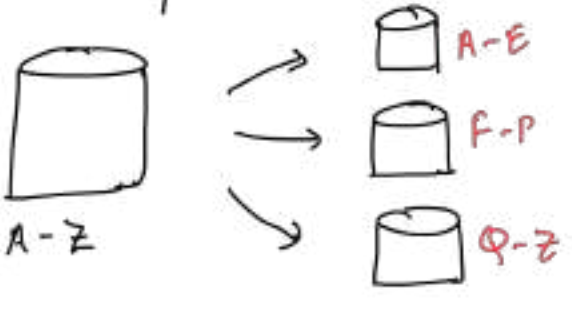
Partitioning of Key-Value Data

↳ Idea is to make sure data is evenly distributed among all nodes.

↳ If data is unevenly stored → called Skewed

① Partitioning By Key Range

Each partition can be assigned at continous range of keys.



ⓐ within each partition, keys are sorted. (SSTable & LSM Tree)

ⓑ range queries are easy.

ⓒ some access patterns can cause hotspots. example Timestamp.

Attach specific prefix to timestamp to solve issue.

② Partitioning By Hash of Key

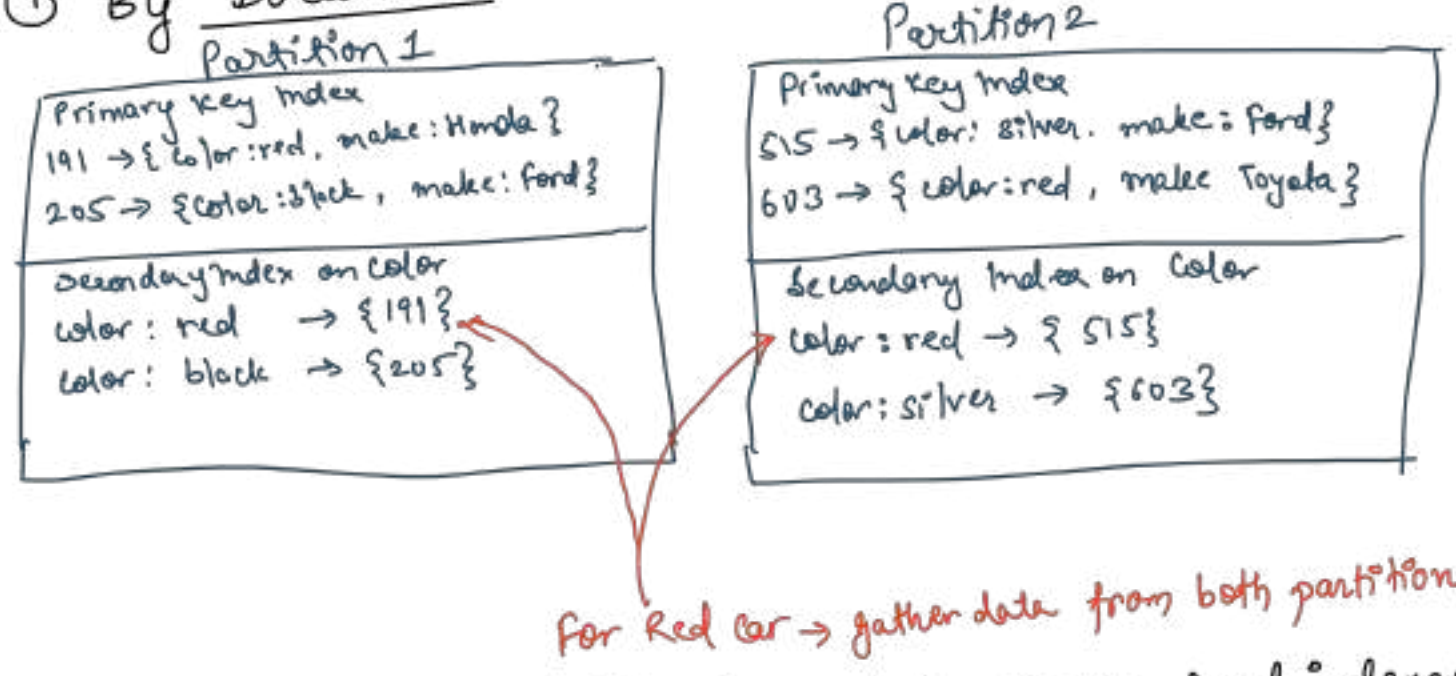
↳ to avoid skew and hotspots.
eg. Cassandra & MongoDB use MD5.

↳ assign each partition range of hashes.
↳ key will be stored in partition whose hash falls in specific range.

Disadvantage
↳ capability of efficient range queries.

Partioning Secondary Indexes

① By Document



→ Each partition is completely segregated, create local indexes.
→ Read queries on secondary indexes could be expensive → gathering data from all partitions.
→ used in MongoDB, Cassandra, Elastic Search

② By Term

↳ construct global index which covers data in all partitions.

↳ global index is also partitioned.
↳ you can use similar to range partitioning.

Benefits

↳ reads are more efficient. → hit only specific partition to gather data.

Downsides

↳ writes can be heavy. → write to single document can affect multiple partitions ↳ terms in doc located on different partitions

↳ example DynamoDB global secondary indexes.

Rebalancing Partitions

↳ add more CPU to handle load.
↳ add more disks.
↳ replace machine.

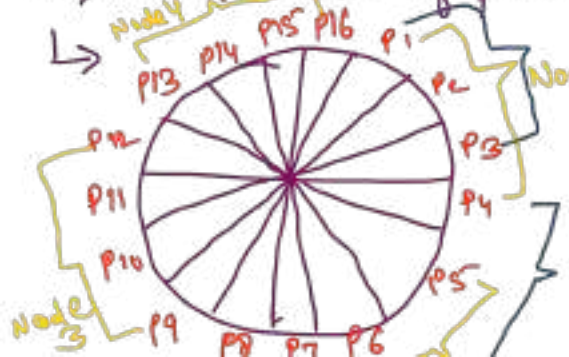
} → requires movement of data from one node to another.

Strategies

① Hash Mod N → not effective approach as number of nodes (N) will change, hence changing node number where data will be stored.

② Fixed Number of partitions → used in Riak, Elasticsearch couchbase, voldemort.

↳ Create lot of partitions and assign these partitions to nodes.



↳ as new nodes are added, few partitions are assigned to node to make distribution even.

→ no of partitions are fixed and only

assignment changes.

→ partition size should be not too small OR not too large.

It's important that you come up with number after proper analysis.

not too small → management overhead

not too large → rebalancing on node failures is expensive.

③ Dynamic Partitioning

↳ Key range partitioned databases (HBase & RethinkDB) create partitions dynamically.

→ as partition grows and exceeds certain limit, it is split into two parts.

→ on data deletion, if required, partitions can be merged

④ Partitioning proportionally to Nodes

In #3 → no of partitions proportional to size of dataset.

In #2 → size of partition is proportional to size of dataset.

In Cassandra → no of partitions proportional to no of nodes.

↳ fix no of partitions per node.

↳ Add new node → partitions becomes smaller as data is moved.

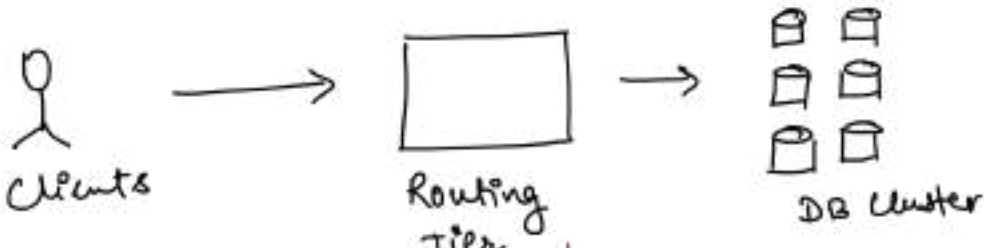
ⓐ randomly choose fix no of partitions to split

ⓑ After split, take ownership of half of them and leave other half in place.

Request Routing → How will you serve read query as rebalancing happens and data moves from one node to another ?

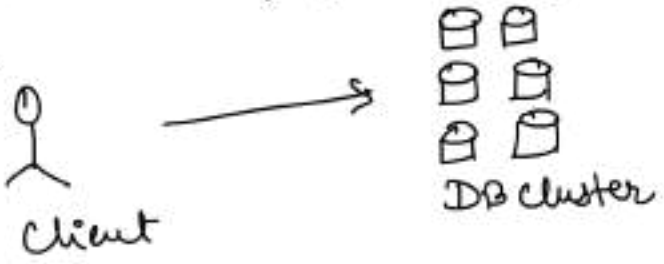
① route request to any node via round robin. If data not found, move to another node.

(2)



determine which node should handle request → partition aware load balancer.

(3) Clients are aware of partitioning and assignment of partitions to nodes.



How will you know assignment of partitions to nodes?

↳ Use coordination Service example Zookeeper.

↳ Ex. HBase, Kafka

Keeps mapping of partitions to nodes.

↳ Use gossip protocol between nodes. eg. Cassandra.

↳ Own routing tier. eg. CouchBase use routing tier called moxi.

Subscribe for more such content.

YouTube Channel - MSDeep Singh

Instagram | LinkedIn | github - msdeep14

Happy Learning ☺

Designing Data Intensive Applications

CHAPTER – 7

Transactions

Youtube channel – MsDeep Singh

Subscribe for more...

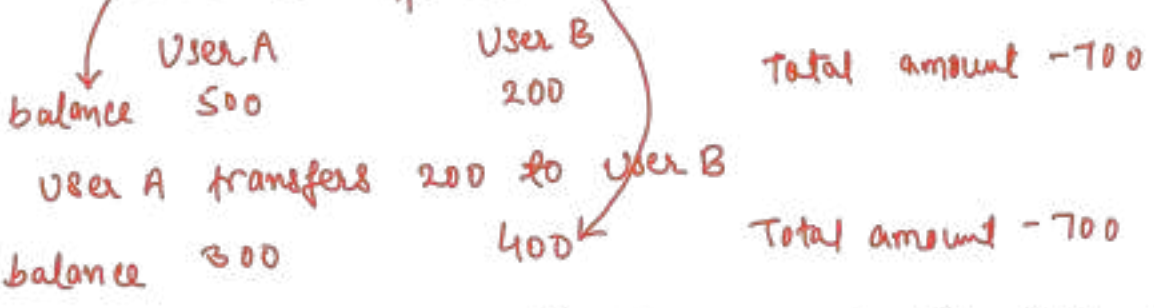
Transaction → group multiple reads and writes into single logical unit.
- executed as single operation
- either all succeeds or all fails.

ACID → Safety gurantees provided by transactions.

① Atomicity → atomic operation
- can't be broken into smaller parts.
- Successful if all parts of transaction are complete.
- for any fault, it will fail.

② Consistency → correctness of database.

Example – Transaction amount between two accounts should be balanced before and after the transaction.



- consistency is more of application property. Atomicity, Isolation and durability are database properties.

application's responsibility to achieve Consistency utilizing A.I.D from ACID.

③ Isolation → Concurrent transactions occur independently in DB without compromising Consistency.

each transaction pretend that it is only transaction being performed

the result will be same if these transactions are carried out sequentially.

④ Durability → once transaction is committed, it will be intact even in case of System failures.

ex - usage of WAL in case of B Trees.

Replication and Durability → No technique is 100% risk free.

↳ we can just take measures to reduce it

Single object writes

↳ you're writing a 20KB JSON to DB, what if network issue after 10KB is written?

↳ Storage engines in-general provide atomicity & isolation at single object level.

atomicity → log for crash recovery - B tree example

isolation → Lock on object.

Multi-Object Transactions

use-cases

① In RDBMS, row in one table has foreign key inference to row in another table.

② multi field updates in document.

③ secondary indexes

Retrying aborted Transaction

① operation completed on server but client didn't get response due to network error.

① If error is due to overload, retry might not help.
↳ use exponential backoff & Jitter.

② Retry for only Temporary errors, not permanent.

Weak Isolation Levels

↳ A reads data and B modifies data at some time.
↳ A and B try to write data to same record concurrently.

Transaction Isolation → System works in manner as everything is working in sequence.
↳ Serializable Isolation.

↳ As strong isolation support has performance cost, some databases tend to use weak isolation levels.

① Read Committed

↳ On Read, you'll see data that is committed. ← no dirty reads
↳ On write, you'll only overwrite data that is committed. ← no dirty writes

use locks { To avoid this issue, transaction B can wait till A is committed or aborted.

→ ① use locks → can lead to more wait time.
→ ② Hold both old and new value for which update is going on.
↳ return old value till new one is committed.

Disadvantage → Read Committed can cause anomaly on read
↳ Read skew
↳ Timing anomaly

Example

Account A	Account B
$500	$500
↳ transfer $100	

Total amount mandeep has in two accounts – $1000

For time being → $400
if read from accounts

$500 ← New value not yet committed

Total amount – $900 ← This issue could occur in case of time on Read.

Solution to above

② Snapshot Isolation → Each transaction reads from consistent snapshot of database.

extremely beneficial for long running read only queries
↑ backups / analytics

How to implement?

↳ write locks to prevent dirty writes.

↳ no locks required on reads.

↳ maintains several versions of objects (unlike 2 in read committed)
↑ Multi version Concurrency Control (MVCC)

Indexes with snapshot isolation

↳ Index point to all versions → index query can filter out object versions not visible to current transaction.

↳ use append only / copy on write B-Tree pages.

↳ don't overwrite pages, create new copy of modified page.

↳ write transaction creates new B-Tree root
↑ old root is consistent snapshot.

Lost Update Problem

① read from db
② modify value
③ write back

} two transactions perform concurrently
↑ one modification could be lost

Solutions

@ Atomic write operations → atomic update operations.
↑ by acquiring lock

(b) Explicit locking → handle locking mechanism in application.
↑ not provided by DB
atomic operations not sufficient
not possible to handle via DB query
ex multiplayer game.

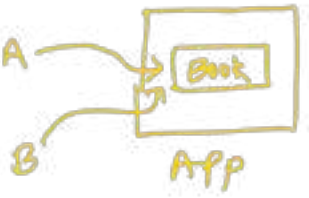
Both users trying to perform action at same piece.

(c) Detect lost updates → allow to execute in parallel and fail if lost update is detected.
concurrently

(d) Compare and set → On update, check the current value is same as what it was read earlier.

Write Skew

Example - Book meeting room.
↳ meeting room should not be booked twice.
↳ Two users tried to book at same time.
↳ check availability → snapshot isolation
↳ room is available, so booked for both?
→ use Serializable Isolation
→ lock multiple rows that are accessed for transaction.



Phantom → write in one transaction change result of search query in another transaction.

Serializability

Serializable Isolation → transactions may execute in parallel but they have same effect as executed sequentially.
↑ no concurrency

(1) Actual Serial Execution → remove concurrency.
↳ no overhead of locking.

② Encapsulate transactions in Stored procedures

↳ transactions should be short.

Book Flight → look from Flight → select → book seat → payment

Single transaction

T1 T2 T3 T4

To support Serial execution, transactions should be short.

to avoid Lag due to locks

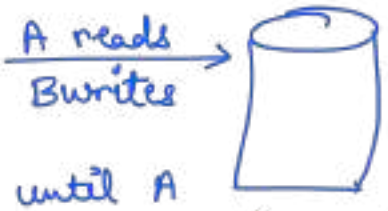
↳ application should submit entire transaction Code to DB ahead of time.

↑ stored procedure.

no I/O wait
no concurrency overhead.

③ Two Phase Locking (2PL)

↳ Several transactions are allowed to write to an object unless no-one is modifying it.



B should wait until A commits/aborts

B wait until A commits/aborts

↑ there might be situation of deadlock if lock is not released. → transaction is aborted by DB in this case.

See you again in next chapter...

Subscribe more more such content.

YouTube - MsDeep Singh

Happy Learning ☺

Designing Data Intensive Applications

CHAPTER - 8

The Trouble with Distributed Systems

Subscribe for such content
YouTube - Ms Deep Singh

Faults and Partial Failures

↳ Single computer in-general works as per written software.
↳ In distributed system, there is possibility some parts are working. → partial failure

Unreliable Networks

Assuming a shared nothing architecture (each machine has its own memory/disk and not accessible to others.), things that could go wrong →

↳ Request is lost / waiting in queue.
↳ Remote machine failed / temporarily unresponsive
↳ Response took more time than expected.

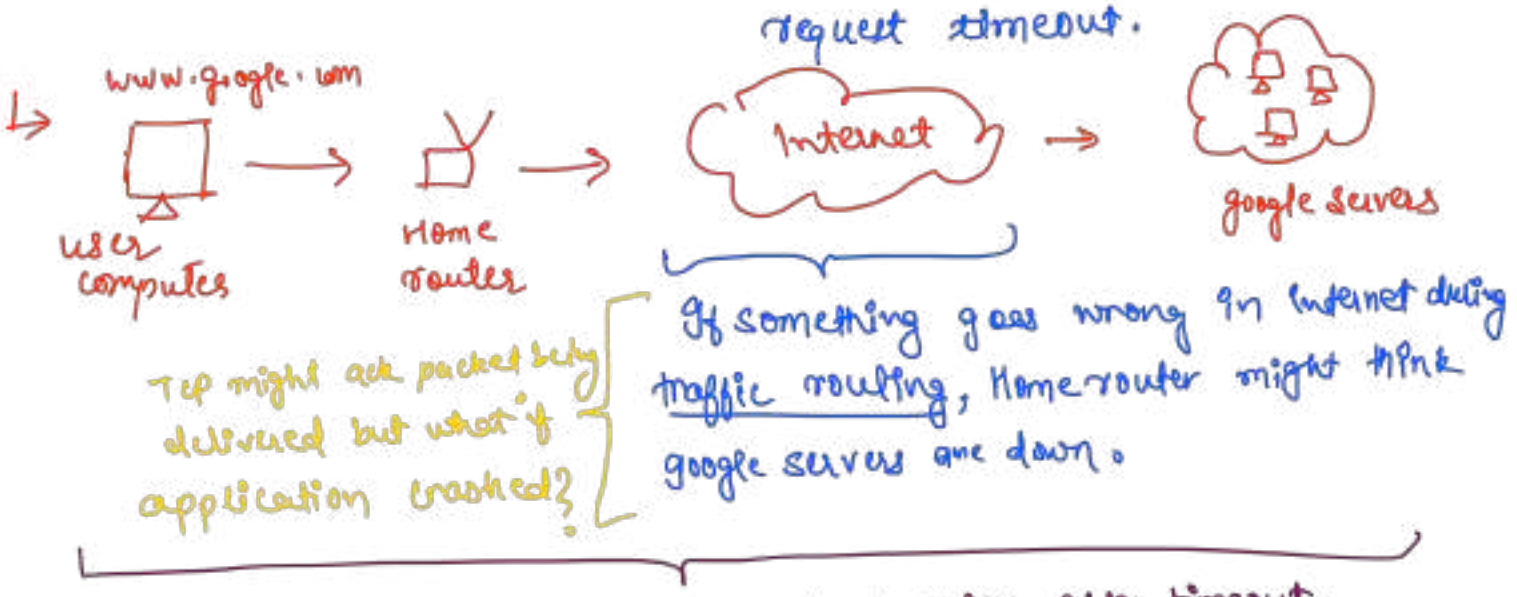
} add timeout so as request don't keep on waiting.

↑ have occurred due to any network faults.

Detecting Network Faults

Automatic detection
↳ Load Balancer don't send request to faulty node
↳ If leader fails in DB, follower is promoted to leader.

What if we're not able to figure out node is dead or not?
↳ eg. Node OS is running but node process crashed.
another node should takeover before



If something goes wrong on Internet during traffic routing, Home router might think google servers are down.

TCP might ack packet being delivered but what if application crashed?

General recommendation is to retry after timeout.
Exponential backoff

Timeouts and Unbounded Delays

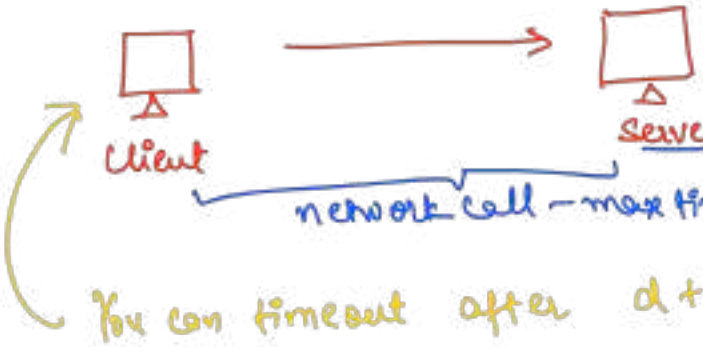
for asynchronous networks, there is no upper limit for packet to be delivered

↳ what should be timeout value?

not too big, not too small.

↳ users have to wait long

↳ node may be actually operational but slow to respond. eg due to overload



You can timeout after $d + r + d = 2d + r$

Network Congestion and queueing

requests more than threshold has to wait to be served.

until the requests are served, they are kept in queue.

queueing can also be at client end

↳ TCP flow control

backpressure

TCP vs UDP

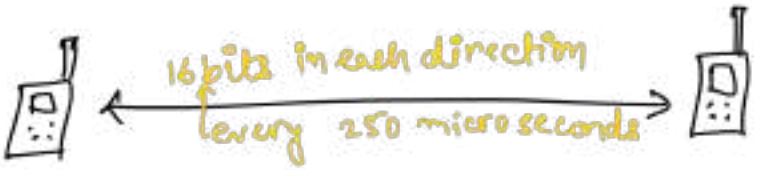
↳ perform flow control
↳ retry – retransmit lost packets

→ client perform #1 and #2
→ we don't want delayed data
eg. live cricket match.

Synchronous vs Asynchronous networks

Fixed line Telephone network

→ fixed guranteed bandwidth is allocated (16 bits)
→ This is Synchronous network.
↳ no queueing - 16 bits reserved
↳ bounded delay → latency is ~fixed.



↳ This concept is not used in TCP

Should we use packet switching over internet?
no → we don't have exact bandwidth requirements.
you'll end up wasting → depends on use-case.
bandwidth or allocating less bandwidth.

Should not be used for bursty traffic.

Unreliable Clocks

↳ Clock is used at multiple places in system. example - Calculate latency of operation.
↳ As there are multiple machines in network, time calculation on each machine might vary.
for synchronization of clocks, mostly Network Time Protocol (NTP) is used

① Time of Day Clocks

↳ current date and time according to some calendar. (wall-clock time)
Example JAVA - System.currentTimeMillies()
↳ return no of second/ms since epoch (January 1, 1970 midnight UTC)

↳ In scenarios of local clock is too far ahead of NTP, it will jump back to previous point in time.
↳ not a good solution for calculation of elapsed time

② Monotonic Clocks → guaranteed to always move forward

JAVA — System.nanoTime()
↳ Not dependent on synchronization between different nodes.
↳ NTP will not cause to jump back or forward.
↳ no synchronization is required.

Example on How NTP can go wrong?

> NTP daemon is misconfigured on servers.
↳ firewall is blocking NTP traffic.

Clock offsets between servers should be monitored.
↳ In case clock on one node drift too much from others → node can be declared dead.
↑ will depend on business your servers are spawned for.
Example - Last write wins algo on DB cluster.

For scenarios where ordering of events is important, Logical Clocks can be used.
why? → based on incrementing counters, It helps in identifying relative order of events.
instead of time elapsed.

Example → How System Pause can cause issues

Database Cluster
single leader per partition

who is leader → A node in coordination can take lease for certain timeout
↑ needs to be updated once timeout expires
How? via synchronized clocks

If nodes don't update lease, it is assumed dead and another node should takeover.

In this perfect setup, what could go wrong?
① lease time depends on time taken from another machine.
what if clocks are out of sync → can cause issues
② To solve #1 - use monotonic clocks
what if there is unexpected pause in System
(a) example due to Garbage Collector (GC)
(b) OS access disk for accessing memory pages (virtual memory)
↳ The thread is paused as I/O takes place
If lot of time is spend in doing I/O rather than actual processing → Thrashing.
③ Can we treat GC as

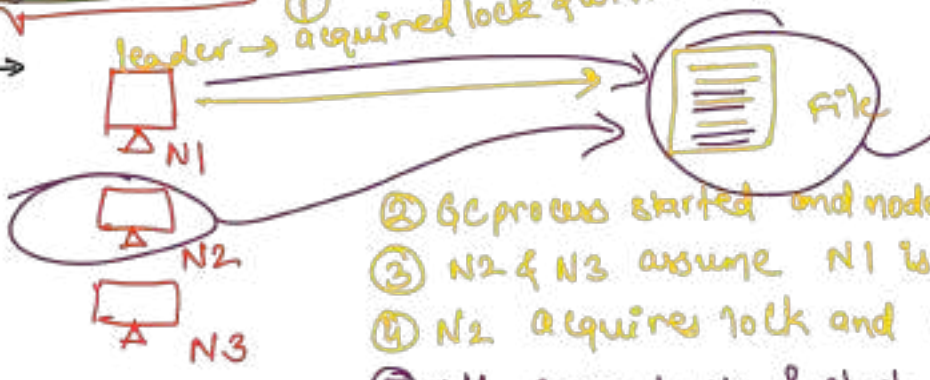
planned outage } → no requests served by node in this time.

② Use GC only for short-lived objects and restart processes periodically.

→ will help in reducing impact on application.

Fencing Tokens

Problem →



② GC process started and node goes to pause state.
③ N2 & N3 assume N1 is dead
④ N2 acquires lock and start writing to file
⑤ N1 comes back & start writing to file. corrupts the file

→ with every lock, keep a counter which is incremented every time a lock is granted for write.

In above scenario

N1 is granted 10
N1 paused
N2 granted 11

→ as N1 tries to write back with token as 10 it is rejected.

what if N1 update the token at its end and send 11 instead of 10.

Byzantine Faults

Safety and liveness → response is sent by server if certain conditions are met.

↳ example - quorum of nodes.

Even if entire system crash / network fail, algorithm should always return correct result.

See you in next chapter...

Happy Learning ☺

Designing Data Intensive Applications

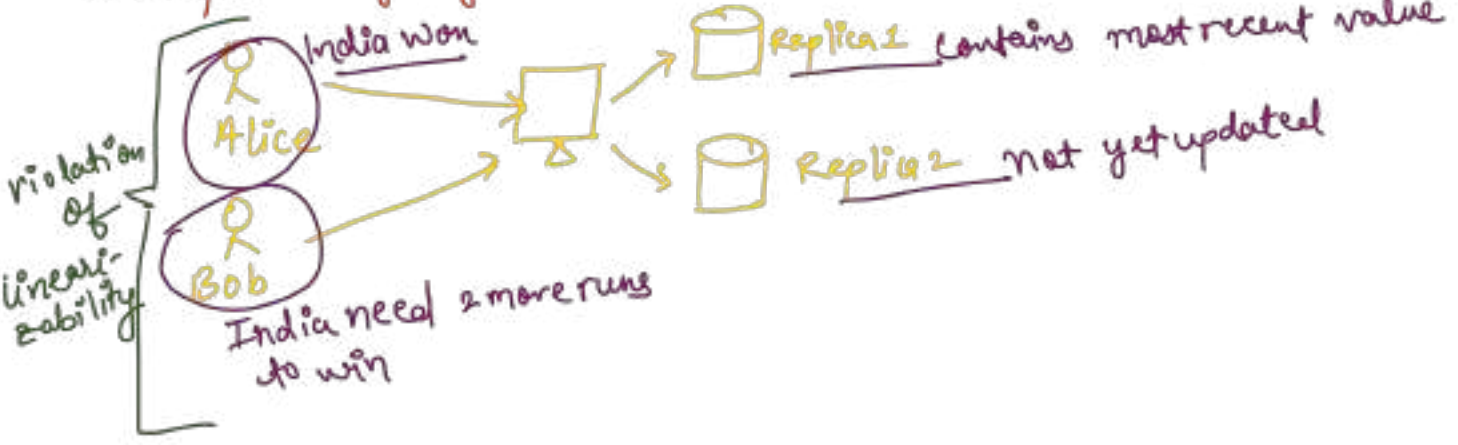
CHAPTER-9

Consistency and Consensus

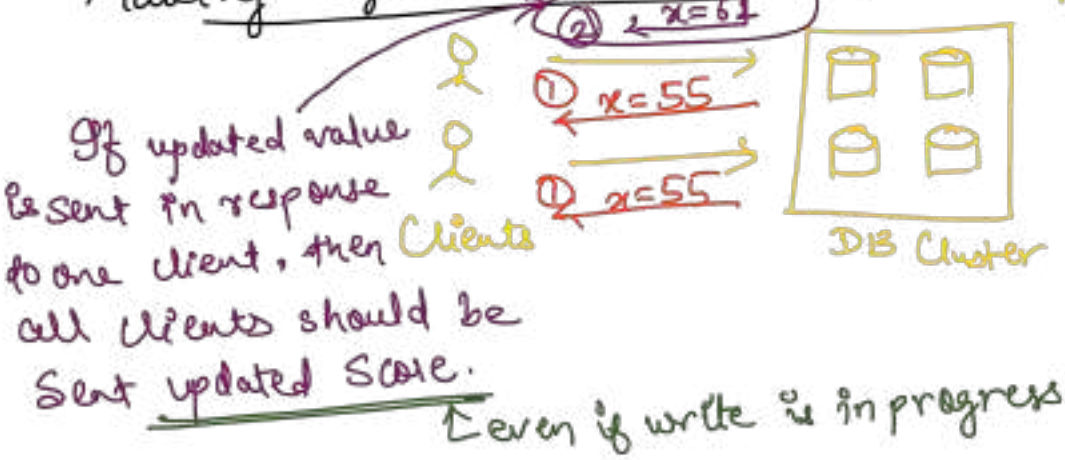
Subscribe for more such content...
YouTube – MsDeep Singh

Linearizability → There could be multiple replicas in database but to client it appears like a single node.
provide strong/atomic consistency.

Example – ongoing cricket match



Making system Linearizable



x → current score of India
x=55
↓
x=61

If updated value is sent in response to one client, then all clients should be sent updated score.
↑ even if write is in progress.

Comparing Linearizability with Serializability

Recency guarantee on reads
Isolation property of transactions

and writes of an object.

Transaction behaves as same if executed sequentially.

why linearizability?

① Locking and leader election

↳ for single leader replication – should only be one leader, not more → split brain

a node can aquire lock on becoming leader.

linearizable. Example zookeeper, etcd.

② Uniqueness Gurantees → Example username should be unique for user.

should be linearizable operation.

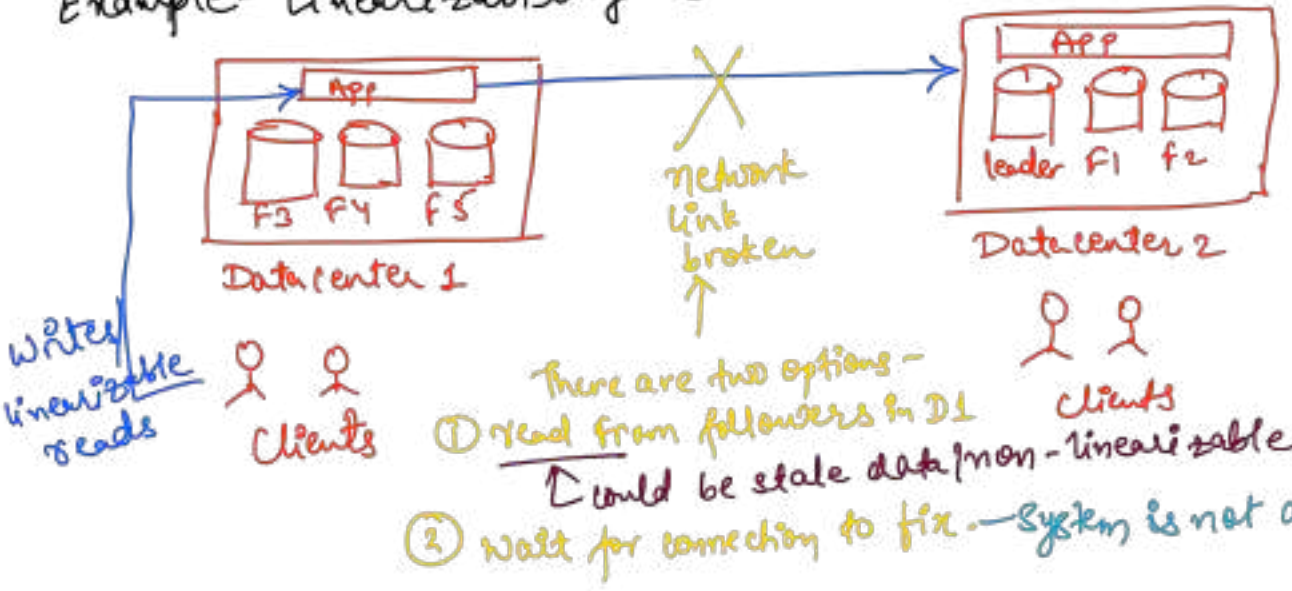
Implementation

↳ Keep only single copy → operation will be atomic by default.

System is not fault tolerant.

↳ Consensus Algorithms → can be used to implement linearizable systems

Example- Linearizability vs Availability. → Single leader replication



There are two options –

① read from followers in D1 – non-linearizable – system is available

↳ could be stale data

② wait for connection to fix. – System is not available

CAP Theorem → Choose either 2 of Consistency, Availability and Partition Tolerance.

Unpractical, CA system is not possible.

linearizability and network delays

↳ It is generally avoided (if our use-case permits) to improve performance.

It is slow even if there are no network faults.

Ordering Guarantees → operations are executed in particular order

① Ordering and Causality

! → ordering helps in preserving causality.

→ Example

@ Consistent Prefix Reads → question should be asked before so as it can be answered. → effect

cause

casual dependency b/w que & ans

Causality impose ordering in events

↳ cause comes first, then effect.

↳ if system obeys this ordering ↳ referred as causally consistent.

Linearizability vs Causality

Linearizability

Total order of operations

allows two elements to be compared. eg. 13 > 5

we can always say which occurred first.

Database abstracts out that there are no concurrent events.

Partial order

the elements can be incomparable or one could be greater than other.

eg. {a, b} and {c, d}

↳ two operations are ordered if they're causally related.

↳ one happened before other.

↳ incomparable if concurrently occurred.

Example Git version control system.

↳ Any linearizable system will preserve causality correctly.

Stronger than causal system

hence performance can be slow.

Sequence Number Ordering

↳ maintaining all causal dependencies could be large overhead.

→ for event ordering, sequence no or timestamps can be used.

↳ from logical clock.

assign seq no to each operation & these no can be compared. → use counters for each operation.

Sequence number for non-causal

example multi leader replication

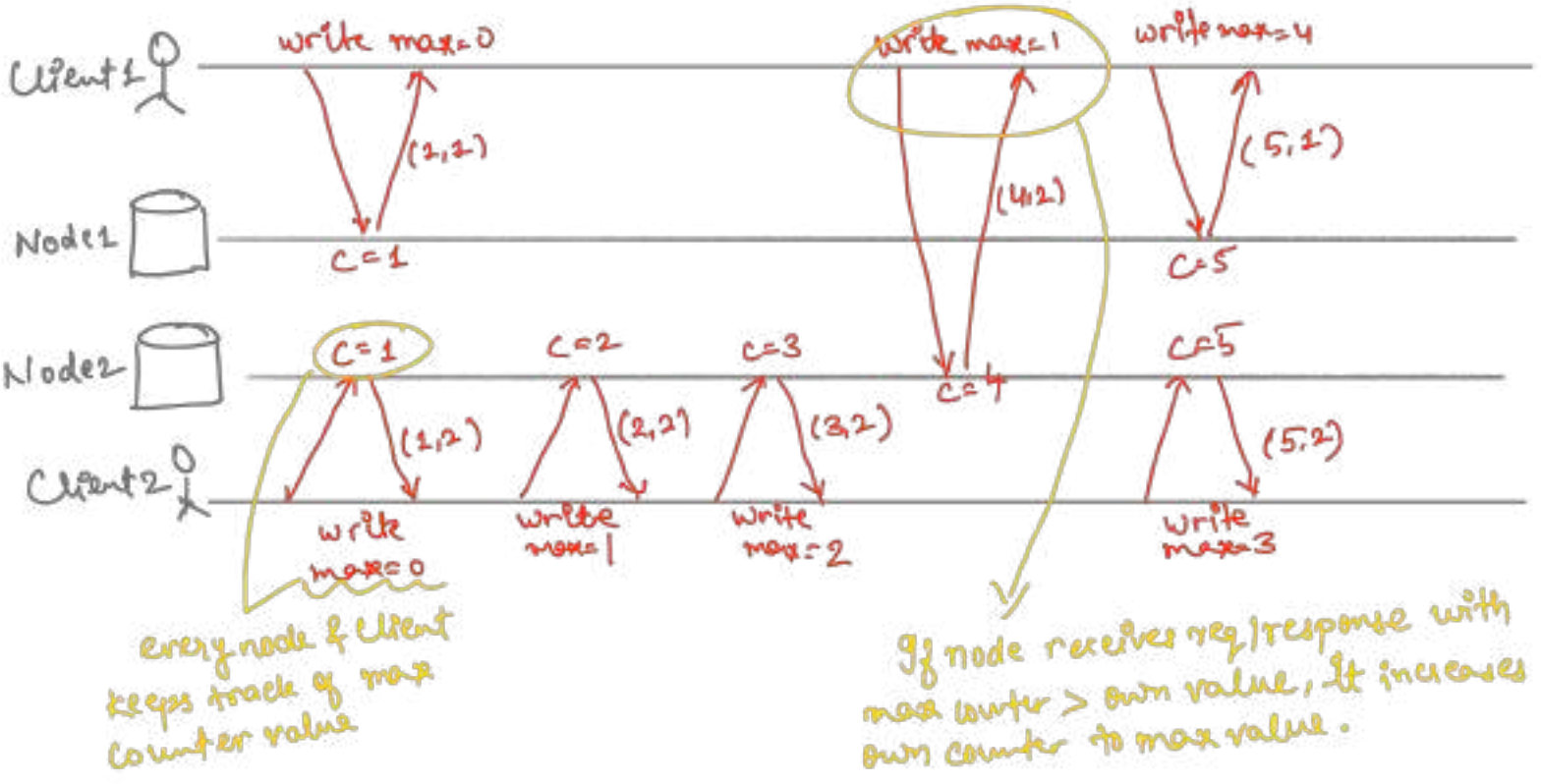
① Each node generate its own numbers.
node should have some identifier so as numbers are unique across clusts
eg one node generate odd and other even.

② Attach Timestamp → used in last write wins algorithm.

③ Assign range of numbers to each node.

→ Generated seq numbers are not causally consistent:
they are not capturing order of events across the nodes.

Solution – Lamport Timestamps {counter, nodeId}
one with greater timestamp is greater
if same time, greater nodeId is greater



The same username problem

Two users trying to create account with same username at same time.
↳ username creation operation is not collected yet. Lamport timestamp will not help.
↳ A node should check with all other nodes if any request with same username.

↑ node has generated operation but don't know what they are?

↑ what if network fault between node communication.

To uniquely identify usernames, total ordering of operations is not sufficient ∑ you should when total order is finalized.

Total order Broadcast.

How can all nodes in cluster agree on same total ordering of operations?

For single leader database? - one node as leader handles this.

What if leader fails?

Total order Broadcast → also used in consensus algorithms.

↳ protocol for exchanging msgs between nodes.

→ Reliable delivery

→ Totally ordered delivery → delivered to every node in same order

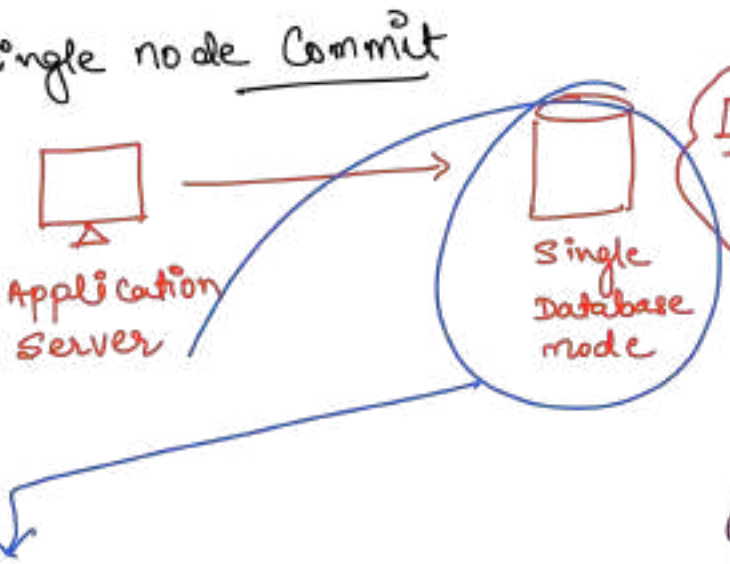
↳ If msg is already delivered, node is not allowed to insert it back in earlier position.

Distributed Transactions and Consensus → several nodes to agree on something

eg. leader selection

Atomic Commit and Two Phase Commit (2PC)

① Single node Commit



On transaction Commit
① Update WAL on disk
② append commit record to log on disk

what if DB crashes before #1 or after #1?
↳ rollback changes

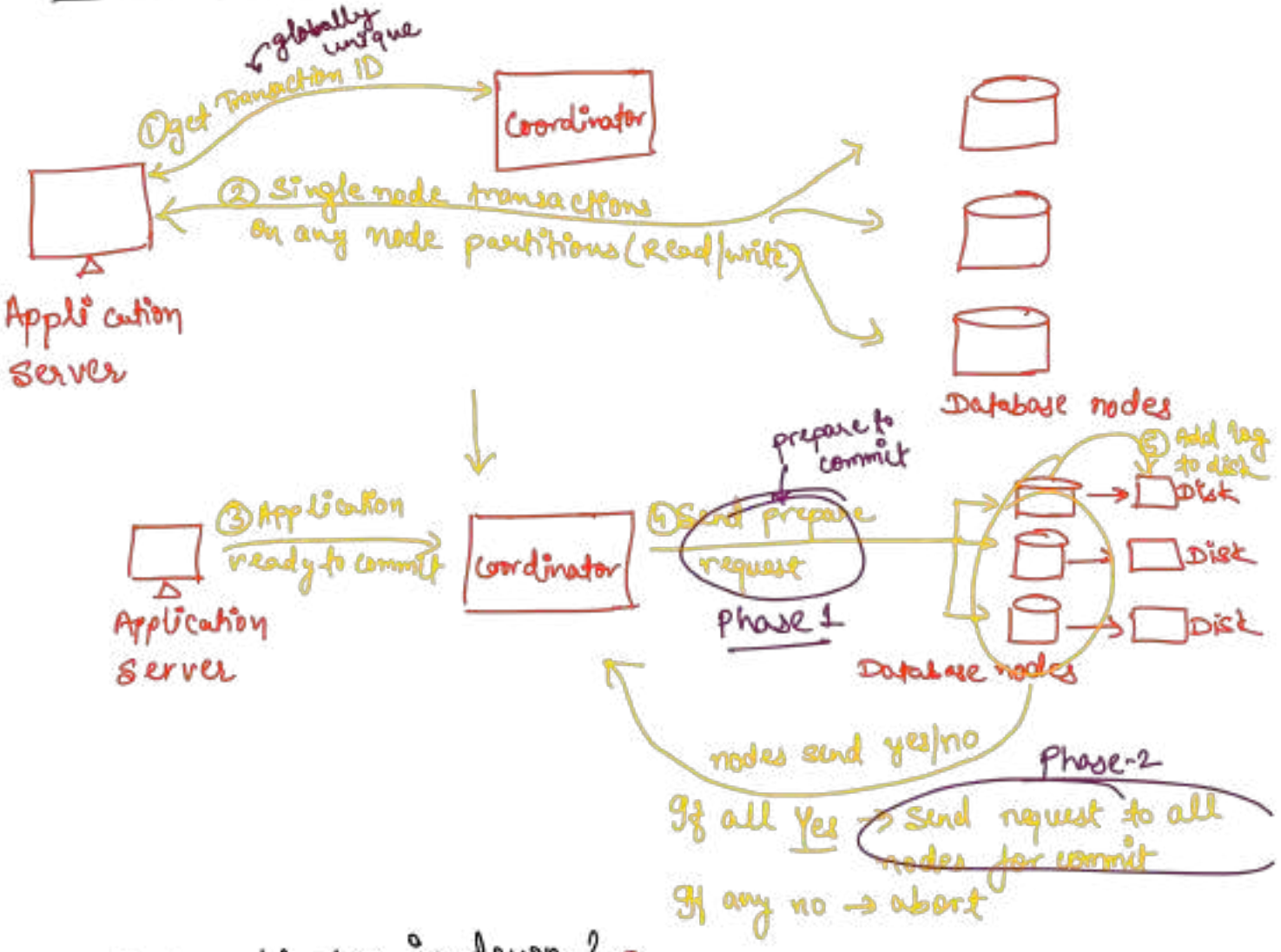
what if crashed after #2?
↳ recover from disk once node restarts.

What if multiple Nodes?

↳ we can't send commit msgs to all nodes. → what if it fails on few nodes? → network delay, node crash, etc.

Difficult to handle.

Two Phase Commit (2PC) → the commit/abort is split into two phases.



what if coordinator is down?

↳ Coordinator is down after sending commit request to one node → other nodes will not know what to do with transaction but wait.

① Either database nodes interact with each to decide what to do with specific transaction. ← not part of 2PC

② Coordinator dumps all actions to disk so it can recover once back online.
↳ what if disk is also corrupt?
↳ manual intervention by admin.

Distributed Transaction Examples

① Database internal → internal transactions among the nodes in cluster.

② Heterogeneous → different technologies distributed transactions
ex: two databases from different vendors, message broker & database

extended Architecture

XA Transactions → 2PC implementation across heterogeneous tech

↳ C API for interfacing with transaction coordinator.

↳ in Java - Java Transaction API → network driver or client library

the driver can send required info to coordinator (prepare | commit | abort)

Fault Tolerant Consensus

safety properties

1. Uniform agreement among nodes → no two nodes decide differently
2. Integrity → node decides only once
3. validity → nodes decides value v, then v was proposed by some node.
4. Termination → every node decides a value that doesn't crash → system should reach a decision even if node crashes

↳ Liveness property

Consensus algo properties

Single Leader Replication and consensus

How is leader chosen? in case of failovers
- manually by operation team
- automatic selection

can split brain problem occur?

→ we need all nodes agree for leader.

↳ consensus

Consensus algo are based on total order broadcast.

like a single leader replication

↳ and it requires a leader

Spiral loop

Solution → Epoch numbering for quorum

→ protocols define epoch number → within each epoch, leader is unique.

If leader dead → election is given an incremented epoch number → monotonic increasing

helpful in resolving conflicts.

Coordination Services eg. Zookeeper, etcd

1. Linearizable atomic operations → for concurrent operation on nodes, only one succeed using atomic compare and set operation.

consensus

② Total ordering of messages
↳ fencing tokens are used to avoid conflicts on lock lease.
↳ Zookeeper use monotonically increasing transaction id & version number

③ Failure Detection → via heartbeats.

④ Change notifications → new nodes added/removed

See you in next video ___

Happy Learning 

Don't forget to subscribe...

Chapter 10 Batch Processing

Book Summary - Designing Data-Intensive Applications

Types of Systems
1. Online Systems — [services] — process request as soon as it's received and send a response
2. Offline Systems — [batch processing systems] — takes large amount of data —> process —> generate output
3. Near-real time systems — [stream processing systems] between online and offline — there might just be a ack response and output sent via events — Chapter 11

Processing data with Unix tools

Example — top 5 popular pages on your website

cat /var/log/nginx/access.log = read the log file

Awk '{print $7}' = split each line into whitespace and output 7th field
Sort = sort alphabetically
Uniq -c = removes duplicate , -c adds a counter
Sort -r -n = sort by number of times page accessed
Head -n 5 = output first 5

Sorting vs in-memory aggregation
1. If dataset has lot of duplicates, it can fit into in-memory hash table
2. Else we can use something similar to SSTables and LSM Trees - combination of in-memory and disk.
3. Unix utility 'sort' automatically spills data to disk and parallelise sorting across multiple cpu cores

Unix Philosophy
1. Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features".
2. Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.
3. Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.
4. Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.

Limitation of unix tools — they run on single machine.

MapReduce and Distributed FileSystems

MapReduce job —
* takes one or more inputs and produces one or more outputs
* Read and write files on distributed file system — HDFS

HDFS —
* based on shared-nothing principle (No special hardware is needed, only computers connected by datacenter network)
* A daemon runs on each node, exposing network service for other node to access files on that node.
* NameNode — keep track of which file blocks are stored on which node
* For failures — replication to multiple nodes

MapReduce Job Execution —

Mapper
1. Take the input record
2. Extract key and value
3. Write output to HDFS

MapReduce framework
1. takes key-value pair from mapper
2. Collect values belonging to same key
3. Write output to HDFS

Reducer
1. Take collection of values as input
2. Produce output . e.g. count of characters

General Details —
1. Number of mappers = number of input files.
2. Number of reducers is configured by person executing the job
3. Framework use hash of the key to Ensure all same keys end up on same reducer
4. Sorting is done in stages (because large data size) similar to SSTables/LSM Trees
5. Shuffle —
    1. Maps writes output to file.

2. Schedular informs reducer work is done.
3. Reducers connect to mappers to download sorted key-value pair for their partition.

MapReduce Workflows

Multiple mapReduce jobs chained together in workflows. Output of one job is used as input to another job.

Example workflow tools — Oozie, Airflow

Reduce Side-Joins and Grouping

MapReduce job is similar to full table scan — read entire dataset. Example — two datasets

User 101 clicked button
User 105 accessed page
User 101 load page
User 102 clicked on hyperlink

Log Entries - user activity

101	Msdeep14
102	Mandeep Singh
105	MsDeepSingh

User Info Table

For collecting data — infer who is user from log entries
1. Access log entry one-by-one and call database —> not efficient
2. Local Cache — Take a dump of database to HDFS where Map-Reduce is running —> Better Approach

Sort Merge Joins

MapReduce Execution
Mapper1 - extract user id (key), activity details (value) from user activity log entries
Mapper2 - extract user id (key), user details (value) from user database

Output —

* the MapReduce framework partitions mapper output by key and then sort key-value pair —> all records (user activity & user database) with same user id become adjacent
* This is used as reducer input
* Reducer can process all records for a user id in one go — keeps only one user record in memory at a time & no network calls.
* Called sort-merge join [Mapper output sorted by key; reducers merge the sorted lists of records from both sides of join]
* The join work is done by Reducers; also referred to as **Reduce-Side Joins**.

The related data is at the same place — helpful in achieving GROUP BY kind of operations.'
e.g. counting page views

What about celebrity user?
Bringing all data at single place for single user will break (creates skew/hot spots) — these records called linchpin objects —
* one reducer doing lot of work as compared to others
* Can take lot of time to MapReduce task completion
* So instead, mappers send the hot key to random reducers based on some hash — parallelisation

Map Side Joins

Mapper reads file from HDFS and writes back to HDFS — no reducers or sorting.

1. Broadcast hash joins
    1. Large dataset joining with smaller dataset (can fit into memory of each mapper)
    2. Essentially, smaller dataset is broadcasted to all mappers
    3. Pig - replicated join; Hive - MapJoin
2. Partitioned Hash Join
    1. Inputs to map-side join are partitioned in same way
    2. e.g. userId%10 — so 10 partitions
    3. Hive - bucketed map joins
3. Map-side merge joins
    1. Partitioned in same way and sorted based on same key

Output of Batch Workflows

Result of data processing?

1. Building Search Indexes
2. Key-value stores as batch process output
    1. Spam filters, anomaly detection, recommendation systems
    2. The output is saved such as it can be queried by some key. e.g. suggested connections for a user on LinkedIn

Comparing Hadoop to Distributed Databases

1. Diversity of storage
    1. Databases are more structured, Hadoop work on files- can be written using any data model or encoding
    2. Dump anything to files and later figure out how to process it - Don't have to worry about schema design
2. Diversity of processing models
    1. MapReduce allows to run your own code on large datasets and don't depend on database software.
    2. Build tools on top of MapReduce as needed. e.g. Hive
3. Designing for frequent faults
    1. If the task is aborted, it can be retried
    2. This enables better resource utilisation — allows tasks with more priority to run prior.

Beyond MapReduce

Writing Map-Reduce jobs from scratch is hard and time-consuming.

Materialisation of Intermediate State

[input] [MapReduce] [output]
[input] [MapReduce] [output]

Output of one job acts as input to another job. Writing out this intermediate state of files is called materialisation.

Issues —
* MapReduce job can only start when preceding job is completed — more execution time
* Intermediate files are replicated on HDFS — overkill for temporary files

To fix these; multiple **Dataflow engines** such as Spark, Tez, Flink — Handle an entire workflow as one job instead of breaking into multiple sub-jobs
* sorting is done only when needed
* No unnecessary map tasks
* Locality optimisations — task consumer and producer on same machine so data transfer over network
* Can write to local disk or memory instead of HDFS — no replication needed

Fault Tolerance
* MapReduce writes intermediate states to HDFS, task can be restarted on another machine on failure
* Spark don't write intermediate state to HDFS; so if machine is lost; data is recomputed

  * For this it keeps track of how data was computed — use Resilient Distributed Dataset (RDD)
* The recomputed data should be same; if not; then entire job should be re-run
* If computation is CPU intensive; materialisation of intermediate states is chapter

Materialisation — as compared to MapReduce, write only needed intermediate states and not all.

High Level APIs and Languages

Writing MapReduce jobs is laborious task - so lot of new platforms came up — hive, Pig, etc.

Writing code - requires more expertise and work
Provide declarative interfaces makes it easy — and internal execution engine can also decide the way to execute for faster task completion.
* Spark, Hive and Flink have cost-based query optimisers — that can change the order of joins and reduce the number of intermediate states.
* Instead of reading entire record from disk, read only required fields

Subscribe on YouTube — @MsDeepSingh

Read my O'Reilly book "System Design on AWS"

Chapter 11 Stream Processing

Book Summary - Designing Data-Intensive Applications

In batch processing systems, the input is bounded.
But the data could gradually arrive over time — unbounded data.

Stream — data is made available incrementally over time. e.g. stdin and stdout of Unix

Transmitting Event Streams

Input - sequence of records or events
* Event is a immutable object containing details of something happened at some of time.
* e.g. order is placed on e-commerce website
* Event is generated by one producer (or publisher) and can be processed by multiple consumers (or subscribers)
* Related events are grouped together into topic or stream

Messaging Systems

* Operates on publish/subscribe model
* Multiple producer nodes can send messages to a topic and allows multiple consumers nodes to receives messages in a topic.

What happens if producers send messages faster than consumers can send them?
* Drop messages
* Buffer messages
* Apply back pressure — blocking sender from sending more messages

Direct messaging from producers to consumers
* Without any intermediary nodes
* UDP multicast in financial industry. e.g. stream stock market feeds
* Web hooks — consumers expose a endpoint and producers can push messages directly.

Issues —
* assumption producers and consumers are online; or application logic should be aware to handle faults
* Consume if offline? It will miss messages

Message Brokers
* message broker or message queue
* Runs a server; producers and consumers connect as clients
* Producers write message to broker and consumers read them from broker
* Durability is handled by broker

Comparison with Database
* database keep data until explicitly deleted
* Data search capabilities such as indexing

Multiple Consumers — multiple consumers read message from same topic
* Load Balancing — each message delivered to one of the consumers, so consumers can share the work.
* Fan-out — each message delivered to all consumers.
* These two patterns can be combined as well

Ack & redelivery
* consumer crash - messages were delivered by consumer but never processed by it.
* Consumer should send a ack that message is consumed
* If broker is unsure of message delivery, it retries on it's end
* If message ordering is important (messages can be reordered in case of retries/redelivery), recommendation is to keep separate consumer per queue.

Partitioned Logs
* Message brokers are built around transient messaging mindset — delete after delivered to consumers
* Databases/Files — delete on user request
* If messages delivered, they can't be recovered — in AQMP/JMS-style messaging

Can we have — durable storage + messaging notification facility???

Using Logs for message storage
Log — append only sequence of records on disk
* this approach is used to implement message broker
* Producer send message by appending to end of log
* Consumer receive it by reading log sequentially
* Similar to unix tail -f

For more scale —
* the log can be partitioned across multiple machines
* Topic can be defined as group of partitions to carry messages of same type

* Each partition, broker assigns monotonically increasing number, (offset).
* Messages within partition are totally ordered
* No ordering guarantee across different partitions
* So , all messages that need to ordered consistently, route them to single partition

Example — Kafka, Amazon Kinesis Streams

Logs compared to traditional messaging
* In general, the broker can assign entire partition to nodes in the consumer group
* Consumer reads messages in partition sequentially, in single-threaded manner.

Downsides —
* number of nodes sharing the work can be almost the number of log partitions in the topic
* If single message is slow to process, it also blocks others

If the downsides overweigh over pros of log based message brokers, use JMS/AMQP

Consumer Offsets
* All messages with offset less than consumer's current offset has been processed
* Broker doesn't need ack for each msg
* Just periodically record consumer offsets
* This helps in increasing throughput

Disk Space Usage
* Keep on appending to log — will run out of disk space
* Log is divided into segments
* Old segments are deleted (or moved to archive store)

Databases and Streams

* The event can also be any operation on the databases — writing to a database. This event can be captured, stored and processed.
* Replication log is stream of database write events, produced by leader

Keeping Systems in Sync

Same data can be stored at multiple places in different forms for different use cases. such as OLTP as source of truth, search index for search capabilities, cache for frequent queries, etc.

* Full database dumps
* Dual writes — application is responsible to write to multiple data sources
  * Concurrency issues
  * One write fails and other succeeds — probable solution, Atomic Commit & Two-Phase Commit

Change Data Capture

Process of observing all data changes in database — dedicated video in System Design playlist

Database — > Parse DB log — > Publish event
e.g. debezium by parsing MySQL binlog

Log Compaction

* Let's say the broker only keeps data for last 14 days. So if a new data source is added built on top of primary database, we need to build new data source on top of some snapshot
* Log Compaction — storage engine periodically looks for log records with same key; removes duplicates; and keep most recent update of key.
* For CDC system, if every change as primary key; and every new update replaces previous value of key; then it's sufficient to keep most recent write.
* So you can rebuild new data source; by reading log-compacted topic from offset zero without taking snapshot.

Event Sourcing

* Stores all changes to application state as a log of change events (similar to CDC)
* But the idea is applied at different level of abstraction
  * CDC — database is used in mutable way; updating or deleting records at will
  * Database don't have to be aware about CDC, it is independent operation
  * Event sourcing, application logic is built on basis of immutable events
  * Event store is append-only
  * Updates/deletes are discouraged or prohibited

Deriving current state from event log

CDC — the entry in database represents current state
Event Sourcing — You need to look all the event history associated with specific entry and then compute current state

* This can be done based on some application logic
* Recomputation again and again can take time
* You can take snapshot or cache is somehow faster response time.
* e.g. series of debit and credit events for your bank account to compute available balance.

Issues —
* if there are so many updates, computing current state is challenging task
* Sometimes data deletion is a requirement so immutability don't serve the purpose. e.g. deletion of data for GDPR compliance

Commands and Events
* Request arrives to the system, it is a command
* System evaluates the request
* If the system decides to process the request, it becomes an event — durable and immutable
* E.g. booking a ticket - command
  * Check if seat is available
  * If yes, generate an event; which can further trigger other systems such as invoicing

Processing Streams

Process the stream once you receive it.
* write to database, cache, search index, etc
* Push events to users e.g. push notifications
* Process one or more input streams to produce one or more output streams; called derived streams
  * The program handling this transformation — operator or a job

Uses of Stream Processing
* Fraud detection system
* Trading systems — price changes in financial market

Complex Event Processing
* short — CEP
* Describe pattern of events that should be detected based on some regex or filtering logic
* If match found, system emits a complex event — with details of event patterns that was detected
* e.g. SQLstream, Samza

Stream Analytics
Aggregations over large number of events
e.g. comparing current statistics to previous time intervals ; such as TPS or latency over 5 minutes interval

Apache Storm, Spark Streaming, Flink, Kafka streams, etc.

Maintaining Materialised views
* Deriving the alternative view onto some dataset that you can query it efficiently
* e.g. search index built on top of database

Reasoning About Time
* Use time windows; such as average over last five minutes

Event Time vs Processing Time
Processing messages may delay —
* Queueing
* Network faults
* Performance issue in message broker; etc

Confusing event time and processing time can lead to bad data. E.g. consumer went down for sometime, once it comes back, it processes the backlog of messages
* Measuring rate based on processing time — there is spike in requests
* Measuring rate based on event time - normal/steady

Which clock?
Assigning timestamps to events is difficult task if events are buffered at several points in the system

For capturing user events based on device clock, the user might have deliberately setup wrong time on device; so we can't trust it.
Solution? Use server time when event was received.
But what if the the server receives event after some time (minutes/hours/days) because the device was offline.
Not a full proof solution.

Solution — log three timestamps
* #1 Event occurring time as per device clock
* #2 Event sent time to server as per device clock
* #3 Event received by server; as per server clock

Offset = #3 - #2. [assuming negligible network latency]
Event occurrence = #1 + Offset

Types of Windows
* Tumbling window
  * Fixed length, every event belong to exactly one window

* Hopping window
  * Fixed length
  * Allows windows to overlap in order provide some smoothing.
* Sliding window
  * Contains all events that occur within some interval of each other.
* Session window
  * No fixed duration
  * Defined by grouping together all events of same user that occurred closely together
  * Window ends; if user is inactive for sometime.

Stream Joins

Join datasets by key

Stream-Stream join (window join)

1. Detect recent trends in searched-for URLs
2. System store query and result returned for every search request.
3. Further, if something for search result is clicked, another event is logged.
4. Join — combine click and search query event ; key = session id

Implementation —

1. Stream processor maintains state. e.g all events in last one hour; indexed by session id
2. For new search/click event, added to the index
3. Stream processor checks other index for events with same session id
4. For matching event; system emit an event saying search result was clicked.

Stream-table join (stream enrichment)

User activity events
Enriching the user activity events from user profile database

1. You can query db for every new user activity event
2. Or keep a copy of database locally to stream processor. — avoid network call

For local copy;
The database can keep on changing as stream is long running
So the local copy can subscribe to database changes — CDC

Table-table join (materialised view maintenance)
For twitter user's home timeline — expensive to iterate over all people user is following ; find their tweets and merge

So build timeline cache; per user-inbox
* user u publishes new tweet; added to timeline of every user who is following u
* For tweet deletion, it is removed from all users' inbox
* User u1 starts following u2; recent tweets by u2 are added to u1's inbox
* Reverse for unfollow action

Implementation —
1. Stream of events of tweets, follow relationships
2. Database with set of followers for each user

Basically joining two tables — tweets and follows

Fault Tolerance
How stream processors can tolerate fault?

Microbatching and checkpointing

* Break stream into small blocks
* Block like miniature batch process; say one second
* Called microbatching
* Used in Spark Streaming

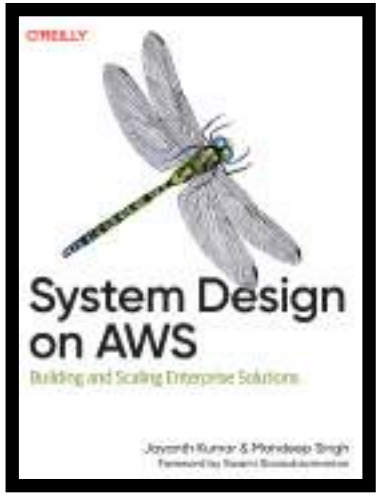
Another approach
* periodically generate rolling checkpoints of state; and write to durable storage
* Used in Apache Flink
* For any crashes, it can restart from recent checkpoint

Atomic commit revisited
Outputs and side-effects of processing an event take effect if an only if the processing is successful

Idempotence
Perform operation multiple times; it has same effect as it is processed once.

Subscribe on YouTube — @MsDeepSingh

Read my O’Reilly book “System Design on AWS”



Chapter 12 The Future of Data Systems

Book Summary - Designing Data-Intensive Applications

Ideas and approaches that can improve the ways we design and build applications.

Data Integration

* Every solution for a problem has pros, cons and trade-offs.
* Appropriate choice of tool depends on circumstances.

Combining Specialised Tools by Derived Data

* A single tool might not solve all the problems so might need combination of multiple. E.g. different datastore for different needs - relational, search indexes, caching, analytics….
  * What one person considers pointless features might be very much important feature for someone else.

Reasoning about Dataflows

* same data is stored in different stores for different access patterns
* You need to be clear about input/output; what's source and destination for the data.

E.g. Database -> CDC -> Search Index
If there is single source of truth for the derived data, it's easier to reconstruct in case of faults.

Derived Data vs Distributed Transactions

Distributed transactions - can be helpful to keep data consistent — generally in sync or atomic operation
Derived data - generally written in asynchronous way

Combining Specialised Tools by Derived Data

* Integration of OLTP db + full-text search index for arbitrary keyword queries
* For simple applications, PostgreSQL full-text indexing feature can be used.
* Search indexes are generally not used as durable system of record.
* One person might consider something as pointless feature but it could be central requirement for someone else.

The limits of total ordering
If system is small, construction of totally ordered event log is relatively easy. Limitations with scale —
* all events should pass via single node which decides ordering. You might need to partition if single machine can't handle the volume.
* If servers are multi-region; network delays can make synchronous coordination inefficient — can lead to undefined event ordering.
* Microservice architecture — two events generated in separate services

Consensus algorithms/total order broadcast work well with single leader node; it becomes difficult in case of multi-nodes

Ordering events to capture causality
* If there is no causal link between events, lack of total order is not an issue.
* Facebook; two users in relationship and broke up
  * User A removed user B from connections and posted something online
  * User B should not see this post.
* If there are separate systems for messaging and friendship; the ordering of events can be lost.
* e.g. message event is processed first and unfriend event later.
* So the notifications can be join between messages + friend list.
  * Logical timestamps can be helpful. The recipients should still have handling for out of order events.
  * Log an event to record state of system and assign unique identifier. This identifier can be used to record causal dependency.
  * Conflict resolution algorithms.

Batch and Stream Processing

* The output of batch/stream processing are derived datasets

Lambda Architecture
1. Record incoming data — append immutable events to dataset
2. Generate read-optimised views from the events
3. This architecture paradigm purposes two systems in parallel
    1. Batch processing systems such as MapReduce (consumes the events and produce corrected version of derived view)
    2. Stream processing system such as Storm (consumes event and produce (approximate) update to the view)

Issues with Lambda Architecture:
* Operational overhead of two systems
* Output from separate systems is combined for user requests — can be hard for complex joinsU

Unifying the Two Systems — need below features
* Ability to replay historical events
* Exactly-once semantics for stream processors
* Tools for windowing by event time

Unbundling Databases

Difference between Unix and databases
* Unix is thin wrapper on hardware resources - files
* High level abstraction, user don't have to worry about implementation details.

Creating an Index
* Take a snapshot of existing data and store them in index in sorted fashion
* Continue on this with any new data added

Meta-database for Everything

A single database can't all solve all the data needs.
* Federated Databases : Unifying Reads — single interface to access all underlying data stores.
* Unbundled Databases : Unifying Writes — synchronising writes across the underlying data stores.

Making Unbundling Work
* Traditional approach to synchronise writes — distributed transactions
  * Every system has different transaction semantics, so distributed transactions could be hard.
  * This might not be a good approach for transactions spreading multiple data stores
* Practical Approach — async event log with idempotent writes
  * Loose coupling between components

Unbundled vs Integrated Systems
* Complexity is high for managing multiple infrastructure (learning curve, operations)
* Single integrated software can be easy to manage.
* If feasible, we can try to use single software for multiple needs as long as it serves the use case. <Don't think of too high scale in beginning, you may never reach there>

Designing Applications Around Dataflow

Separation of Application code and State
* Application code is deployed with tools such as Docker, Kubernetes, etc.
* Database is used as durable storage
* Most web applications are stateless — easy to add/remove servers

Dataflow : Interplay between state change and application code
Application code can respond to state changes — by consuming events

Stream Processors and Services
Example — customer is buying something priced in USD but paying in INR; you need currency conversion logic
* Microservices — query exchange-rate service/or database to get currency rate
* Dataflow approach — subscribe to stream of exchange rate updates and keep record of exchange rate in local database.

* dataflow approach is faster
* Dataflow is more robust, independent of failures of another service

Reads are Events too
Generally write requests are seen as events but you treat reads as well as events
* read event goes to stream processor
* Processor emits the results of read to output stream.

Multi-partition Data Processing
* For queries touching single partition, using streams for reads could be overkill
* Could be useful for complex queries needing multi-partition data

Aiming For Correctness

Databases
Just because database has safety properties such as serialisable transactions; the issue can still happen if there is bug in application (write incorrect data or delete some records)

Exactly-once execution of an operation
* Processing twice is a form of data corruption.
  * Idempotency is one way to overcome this.

Duplicate Suppression
Suppress duplicates
* TCP use sequence numbers to figure out lost/duplicated packets.
* Retransmit lost packets and remove duplicates.
* e.g. database connection - transaction is tied to client connection
  * Network issue at client after sending COMMIT and before getting response from db server
  * Client can’t know if transaction commited/aborted
  * Client can retry but it’s outside of scope of TCP duplicate suppression (because new connection)

* Two-phase commit is solution to 1-1 mapping b/w TCP connection & transaction.
* Is this enough ?
  * No
  * What if connection issue between user and application server
  * No response received by user
  * So user retry
  * So duplication can still happen

Uniquely identifying requests
Consider end-to-end flow of request.
* generate unique identifier such as UUID
* This id is used by all the components involved in serving the request

Applying end-to-end thinking in data systems
You need end-to-end solution to fix the problem;
One component handling it really well don't solve the problem

Doing The Right Thing

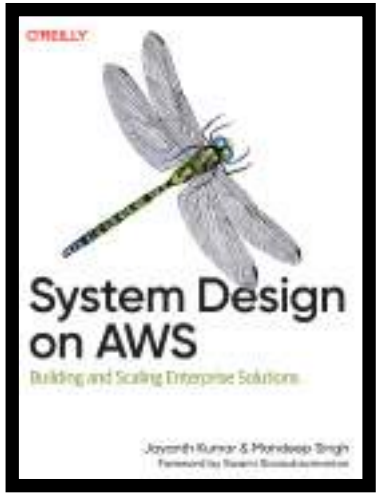
Every system is built for a purpose — and every action has consequences (intended and unintended)

*** Predictive Analysis**
Data analysis to predict weather
* **bias and discrimination**
* **Feedback loops**
* **…**

Privacy & Tracking
* **Consent**
* **Privacy and use of data**
* **Data as assets and power**

Subscribe on YouTube — @MsDeepSingh

Read my O’Reilly book “System Design on AWS”



📖 Passionate about System Design and want to learn how to build large-scale systems using AWS? "System Design on AWS" is the ultimate resource for you. Get your copy -
📕 O'Reilly: https://oreil.ly/ruQbc
📕 Amazon.in: https://amzn.to/4jExkte
📕 Shroff Publishers: https://www.shroffpublishers.com/books/9789355428035/
📕 Amazon.com: https://amzn.to/3D3BIBJ
📕 DragonflyDB (Free chapters): https://www.dragonflydb.io/content/aws-system-design