Designing Data Intensive Applications

CHAPTER 1

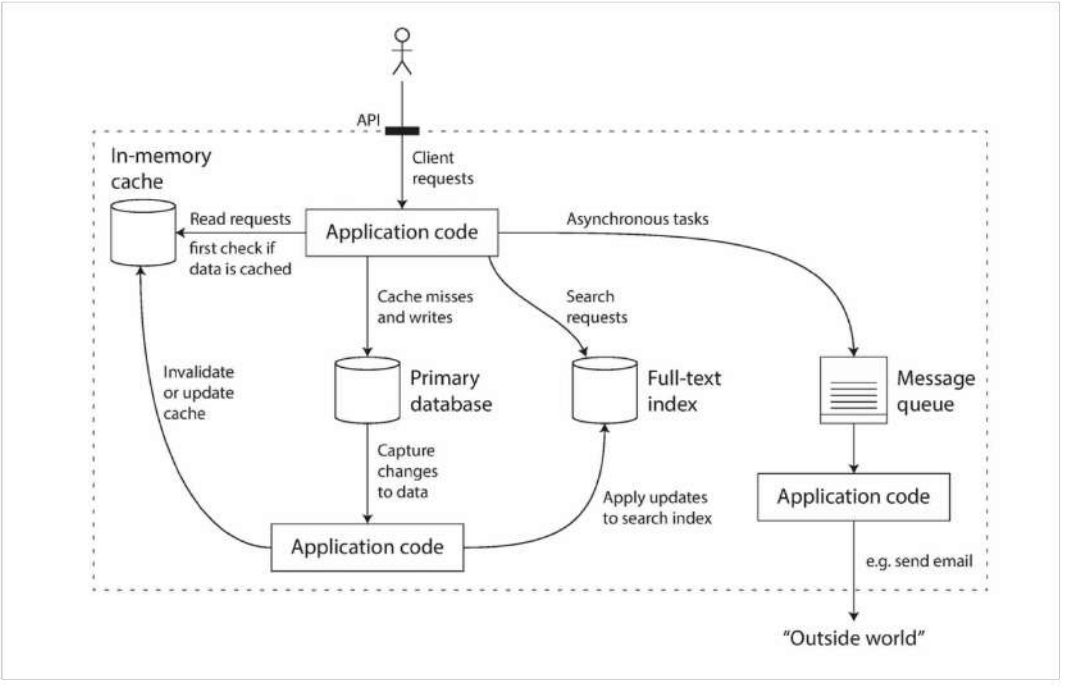
Reliable, Scalable and Maintainable Applications

Reliability → Tolerating Hardware and Software failures
Human Error.

Scalability → Measuring load and performance
Latency percentiles, throughput

Maintainability → Operability, Simplicity and evolvability

One Possible Architecture



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.

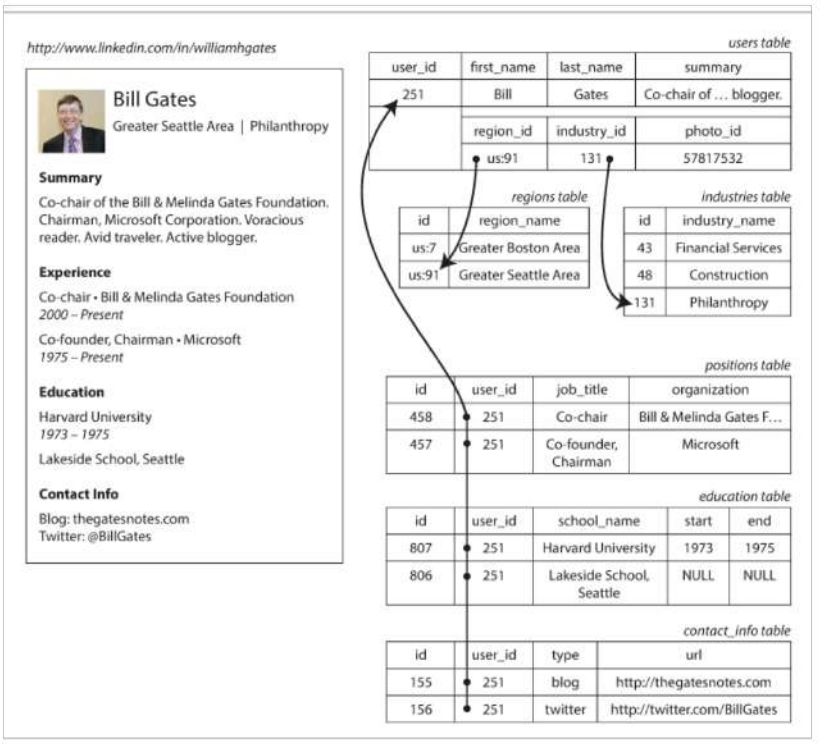
Why NOSQL?

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.

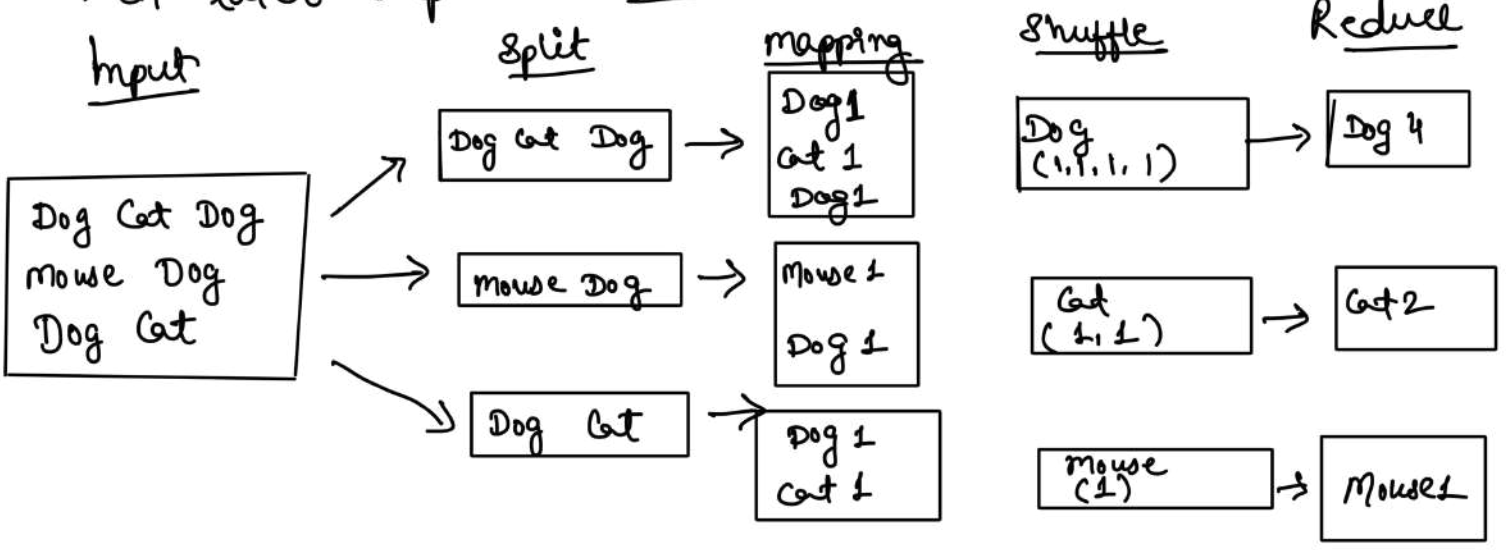
ject

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.

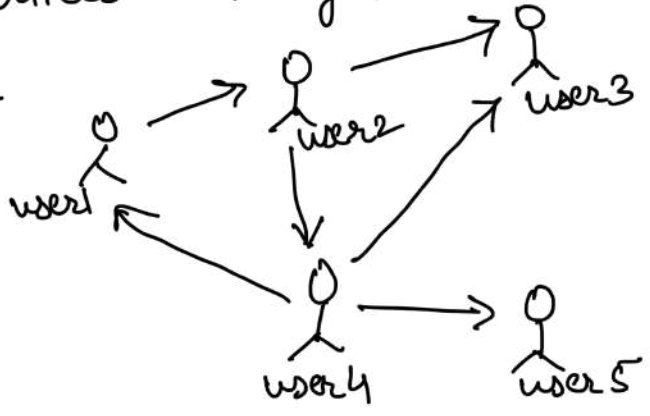
↳ many-many relationships can be very well handled by RDBMS.

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

∑ Then graph data model is more suitable.

Graph → vertices and edges.

Example



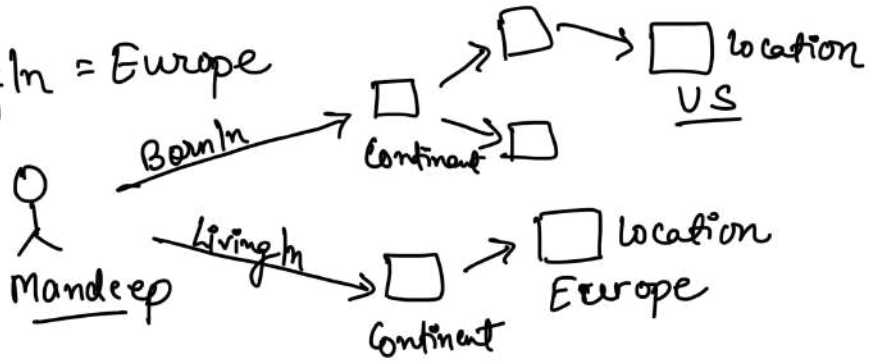
↳ Graph provides flexibility in data Modelling as compared to RDBMS.

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.

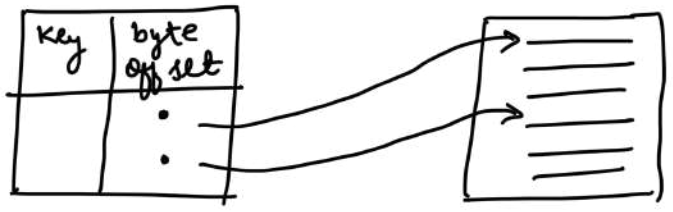
Hash Indexes → How will you index data on disk?

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 Bitcask (Storage engine in Riak)

↳ In memory Hashmap 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 hashmap.

↳ 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)

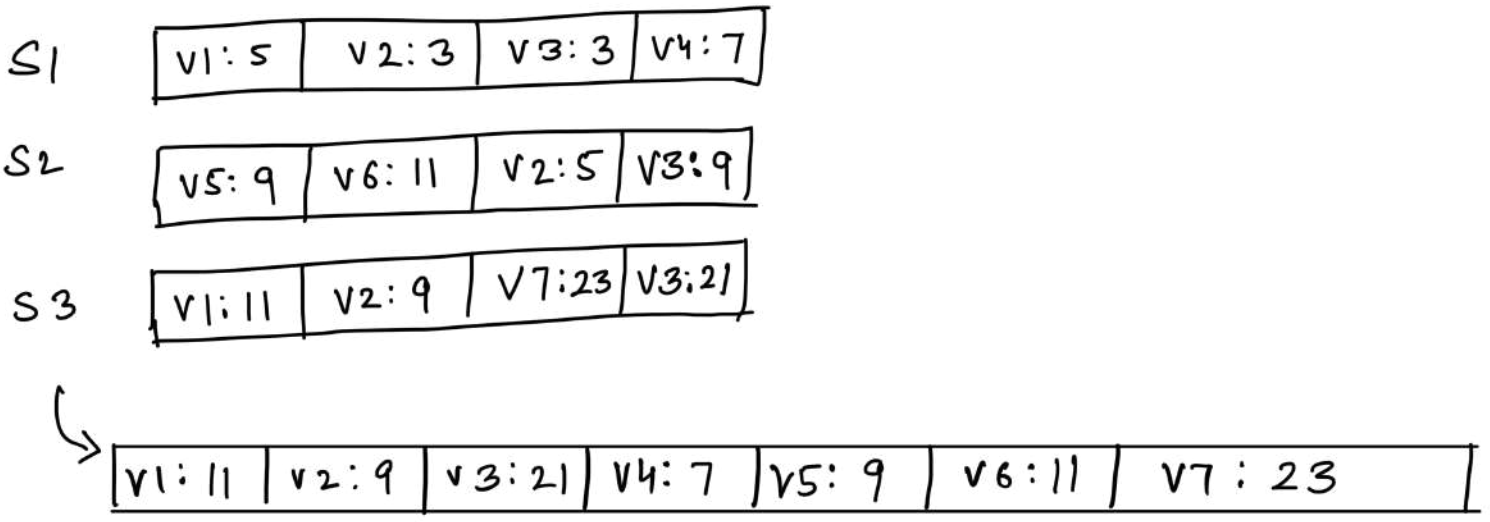
(A) Merging of segments → algorithm similar to merge sort.

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.

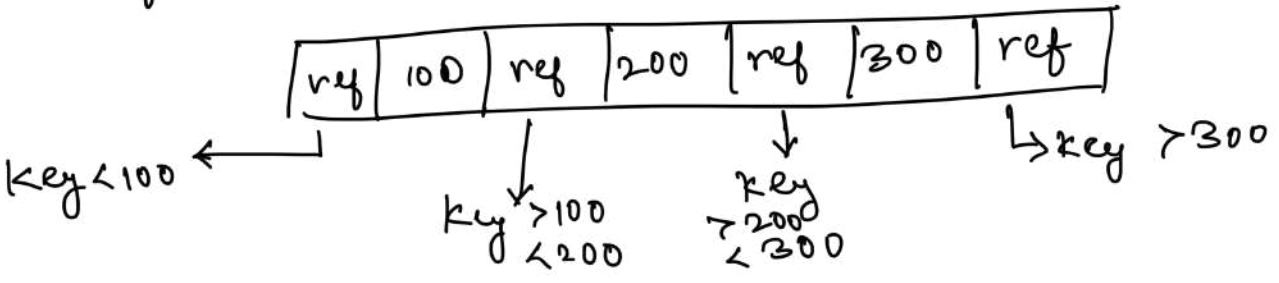
Performance of Database built on LSM Tree

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 existing 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

① Read - small number of records per query fetched by key

② write - Random access, low latency writes

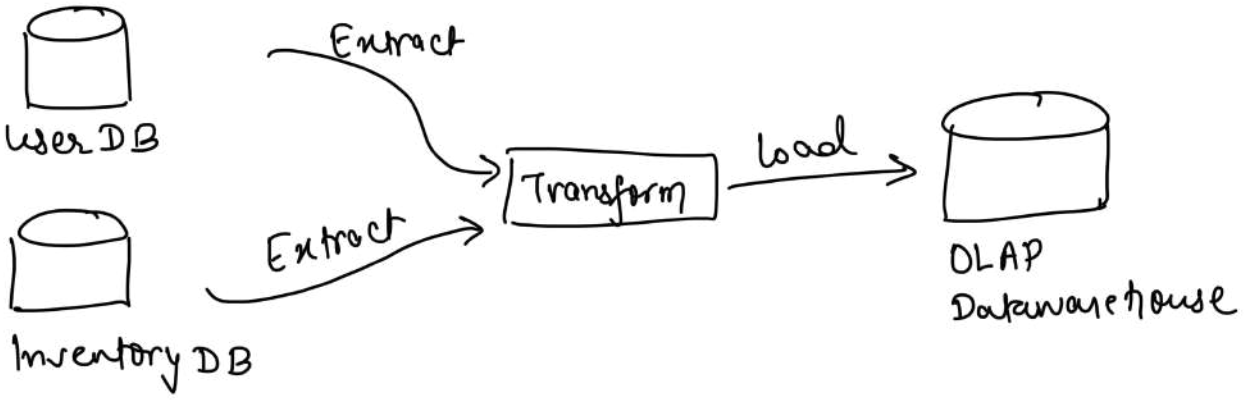
③ GB to TB

OLAP

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 columns to RDBMS

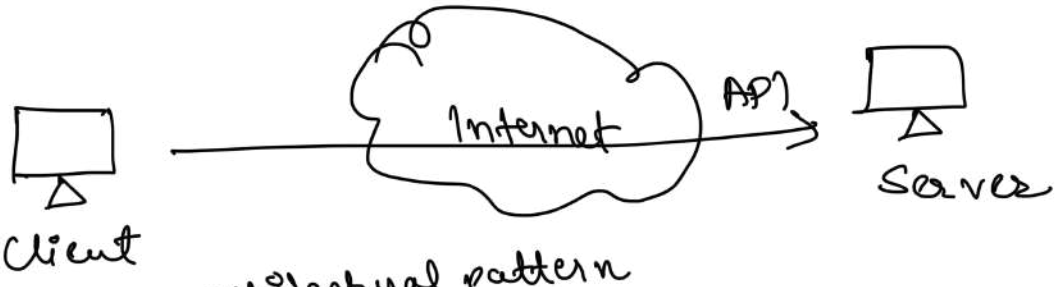
↳ You can specify a default value for newly introduced columns.

↳ 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.

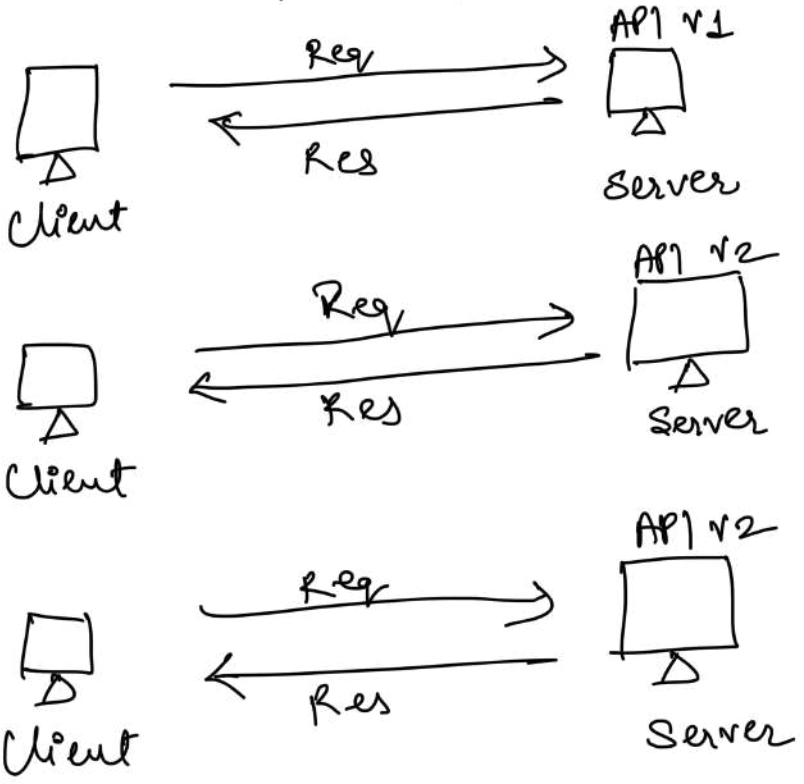
eg. GET, PUT

RPC → Remote Procedure call

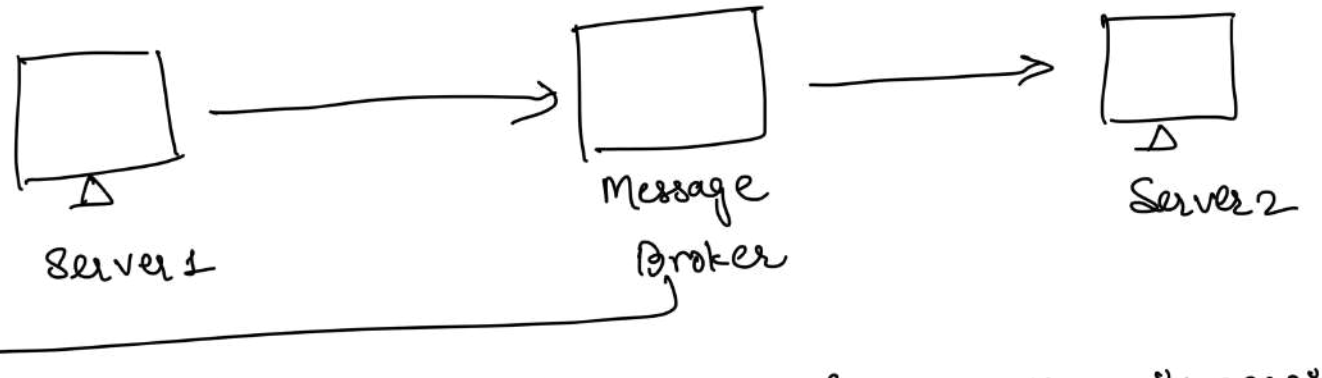
↳ make request to remote network service.

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 receivers.

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

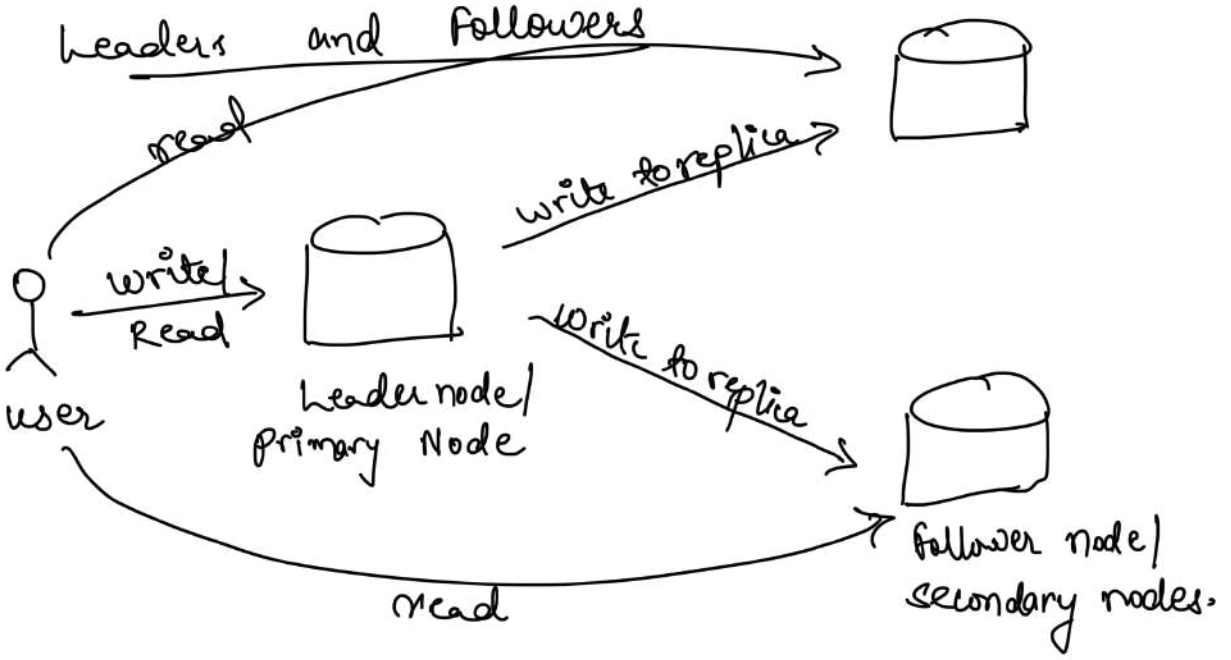
Replication

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.

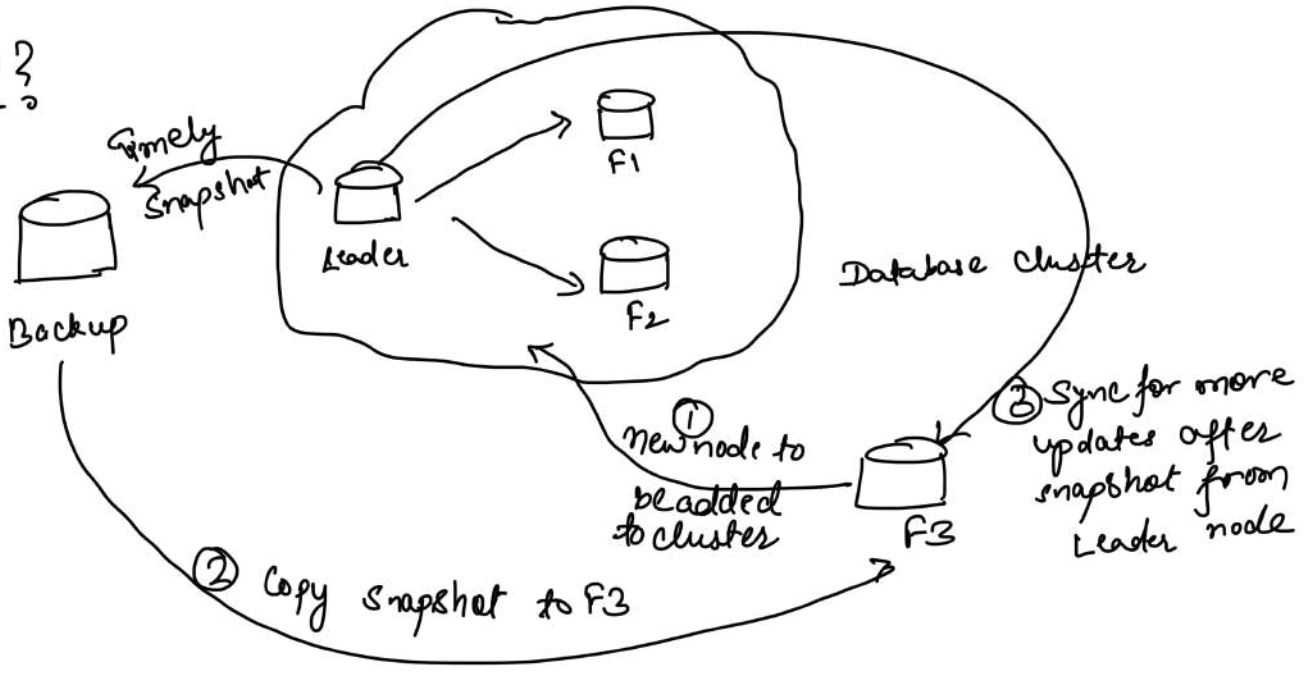
④ for write → leader node should be available.

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.

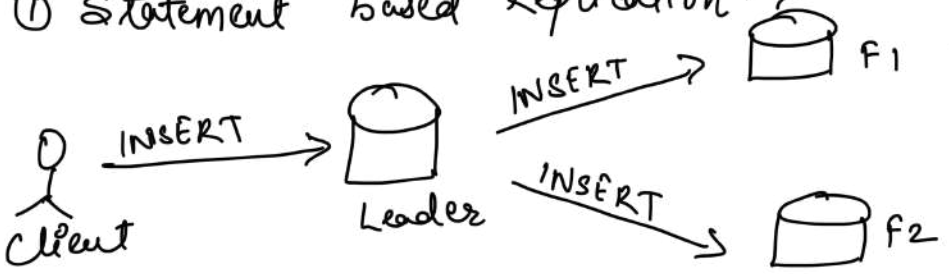
ⓒ System use new leader → new leader accepts writes from clients

↳ 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.

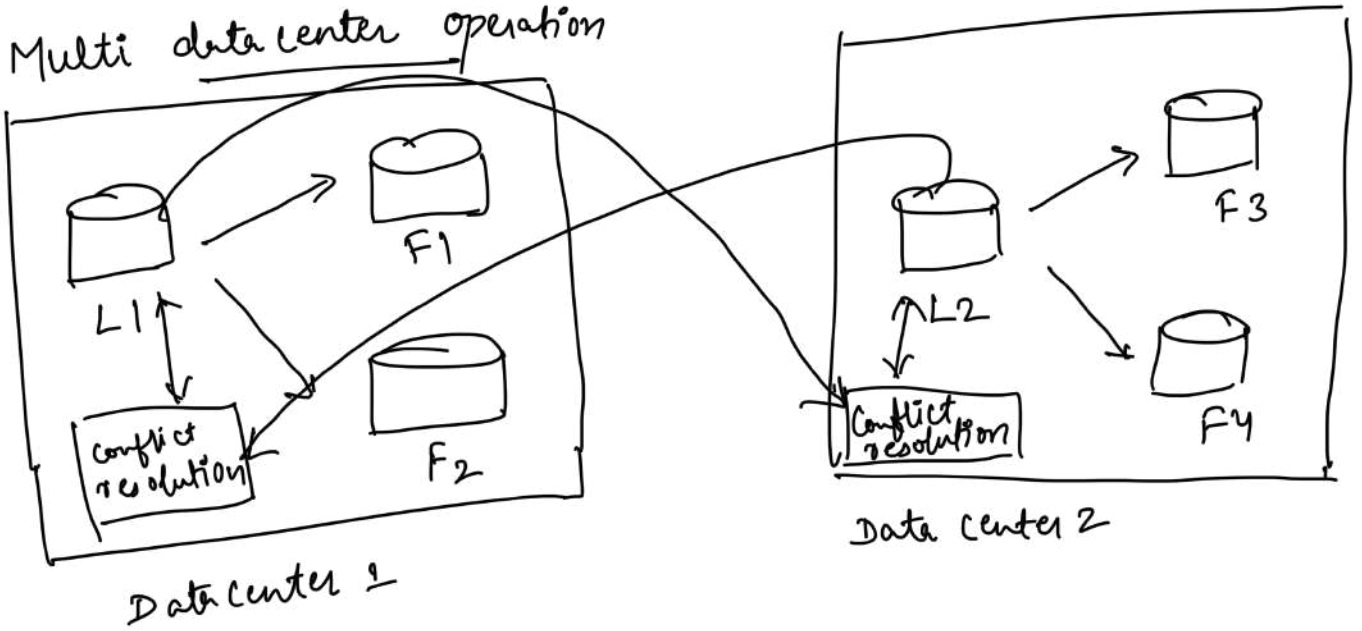
⑤ for 1minute after write, read only from leader.
what if same user reads info on multiple devices?
↳ can user requests routed to same datacenter to handle this.

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.

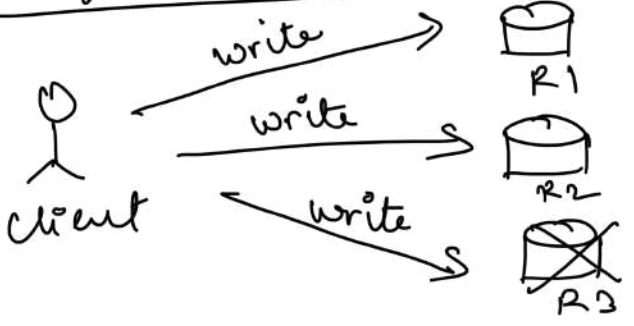
③ Custom Application Code

↳ 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

$n \rightarrow$ odd number
$w, r \rightarrow (n+1)/2$

① If you set lower values of r and w ($r, w <= n$) → 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 timestamp 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

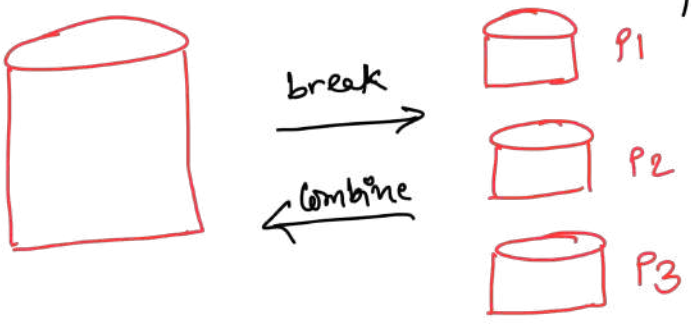
Happy Learning ☺

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

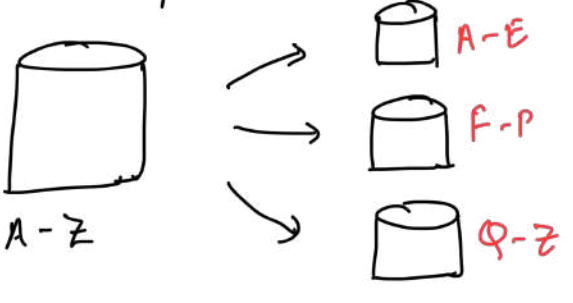
Partitioning of Key-Value Data

↳ 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

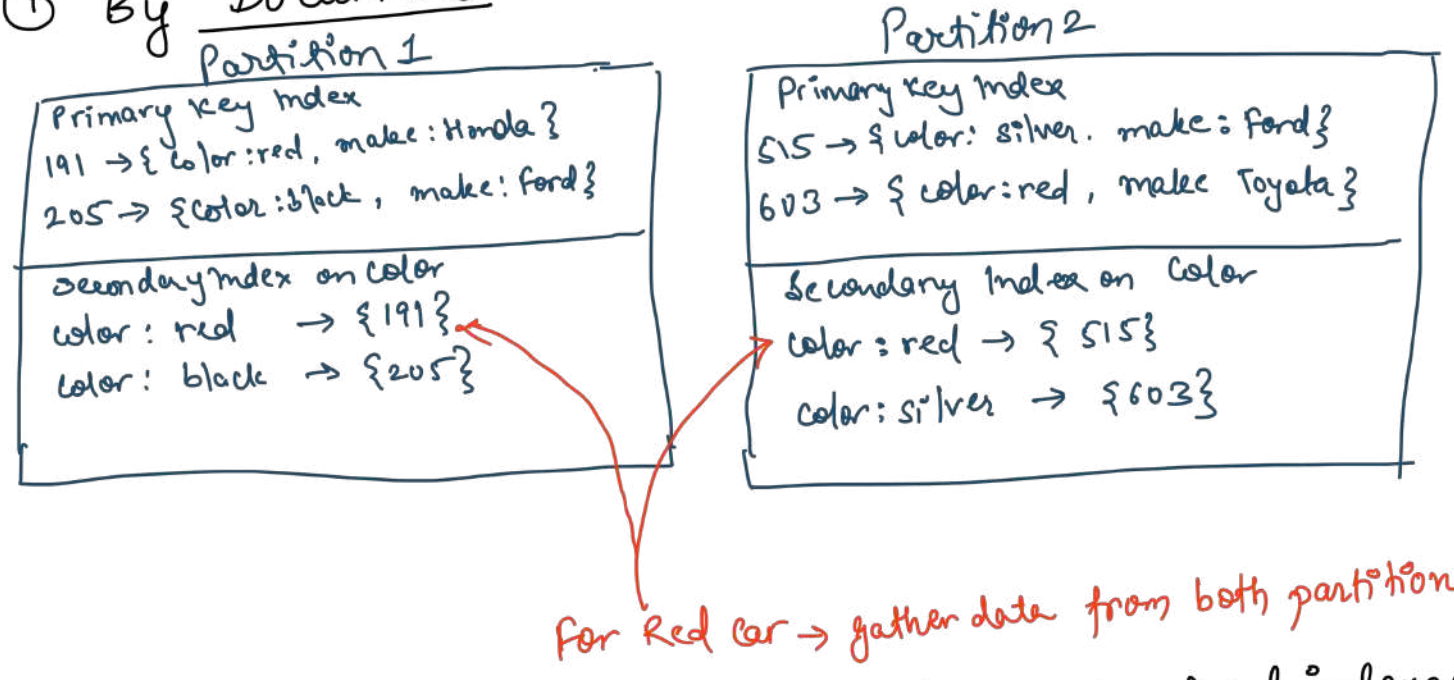
↳ to avoid skew and hotspots.
eg. Cassandra & MongoDB use MD5.

↳ 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.

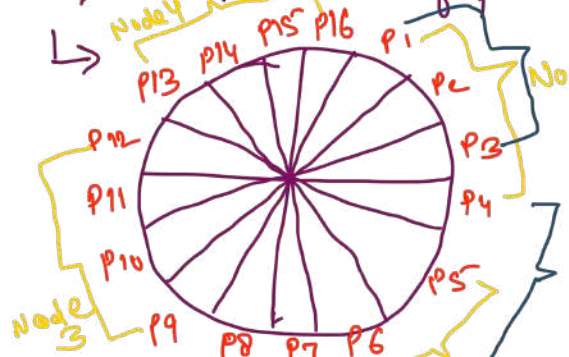
} → requires movement of data from one node to another.

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.

management overhead

rebalancing on node failures is expensive.

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

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.

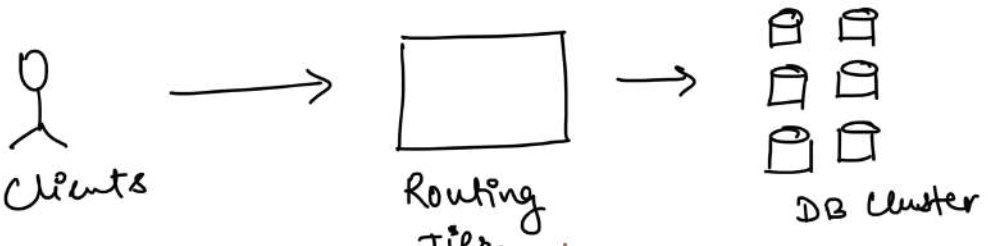
↳ Add new node → partitions becomes smaller as data is moved.

ⓐ randomly choose fix no of partitions to split

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

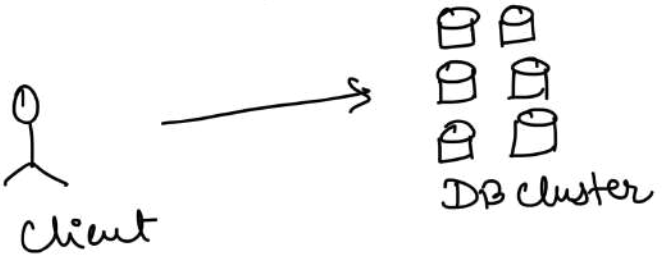
Request Routing → How will you serve read query as rebalancing happens and data moves from one node to another?

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...

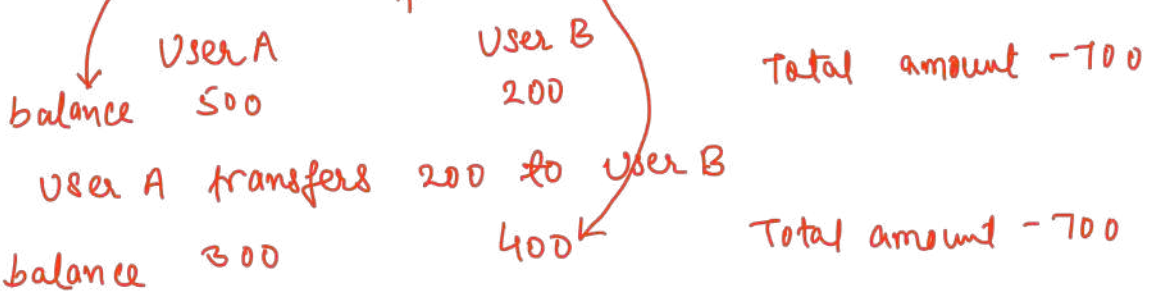
Transaction → group multiple reads and writes into single logical unit.
- executed as single operation
- either all succeeds or all fails.

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 responsibity 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.

④ If error is due to overload, retry might not help.
↳ use exponential backoff & Jitter.

③ 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	

For time being → $400 if read from accounts

Total amount mandeep has in two accounts – $1000

$500 ↳ New value not yet committed

Total amount – $900 ↳ this issue could occur in - 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

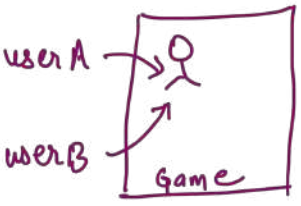
① read from db
② modify value
③ write back

} 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.



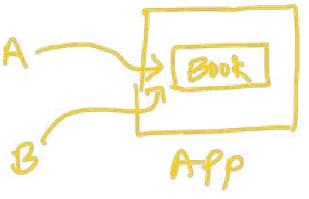
Both users trying to perform action at same piece.

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

(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

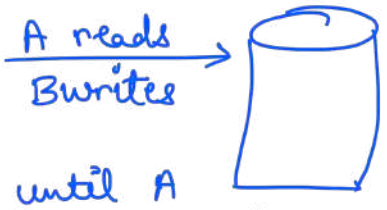
↳ application should submit entire transaction code to DB ahead of time.

↑ 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

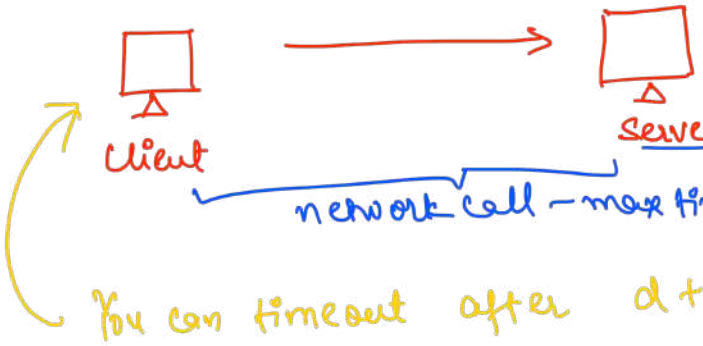
for asynchronous networks, there is no upper limit for packet to be delivered

↳ 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

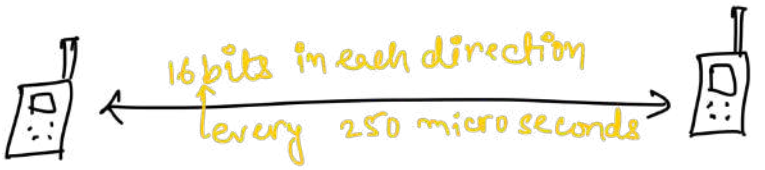
↳ perform flow control
↳ retry – retransmit lost packets

→ dont 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.

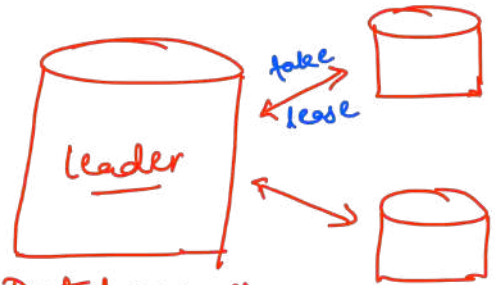
Example on How NTP can go wrong?

> 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

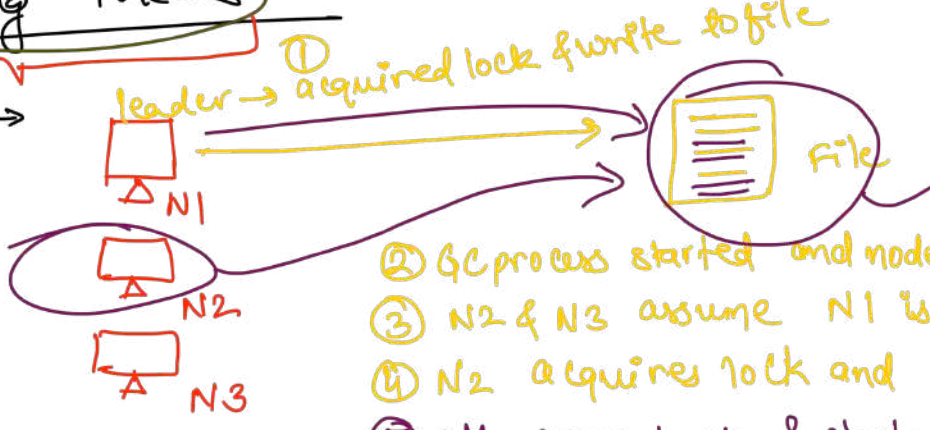
planned outage } → no requests served by node in this time.

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

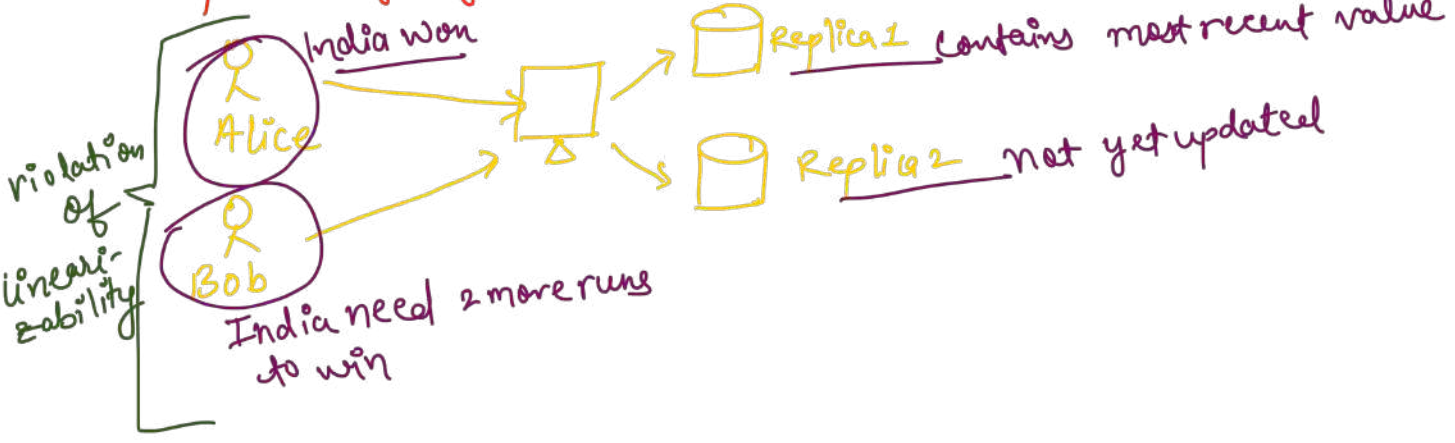
Subscribe for more such content...

YouTube – MsDeep Singh

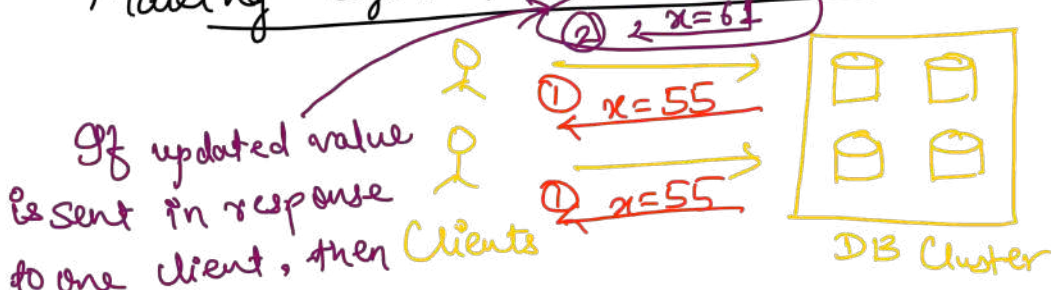
Linearizability → There could be multiple replicas in database but to client it appears like a single node.

provide strong/atomic consistency.

Example – ongoing cricket match



Making system Linearizable



If updated value is sent in response to one client, then all clients should be sent updated score.

↑ even if write is in progress.

x → current score of India

x=55 ↓ x=61

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.

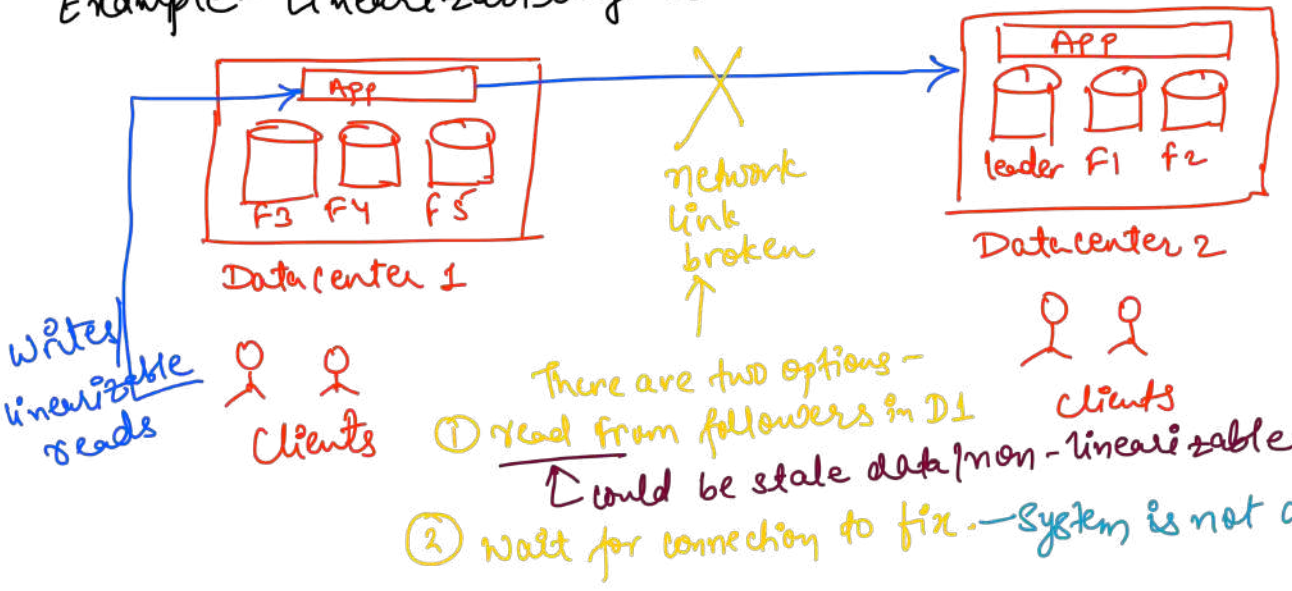
Implementation

↳ 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 ∴ could be stale data/non-linearizable – system is available

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. (cause → effect; 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.

Causality

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

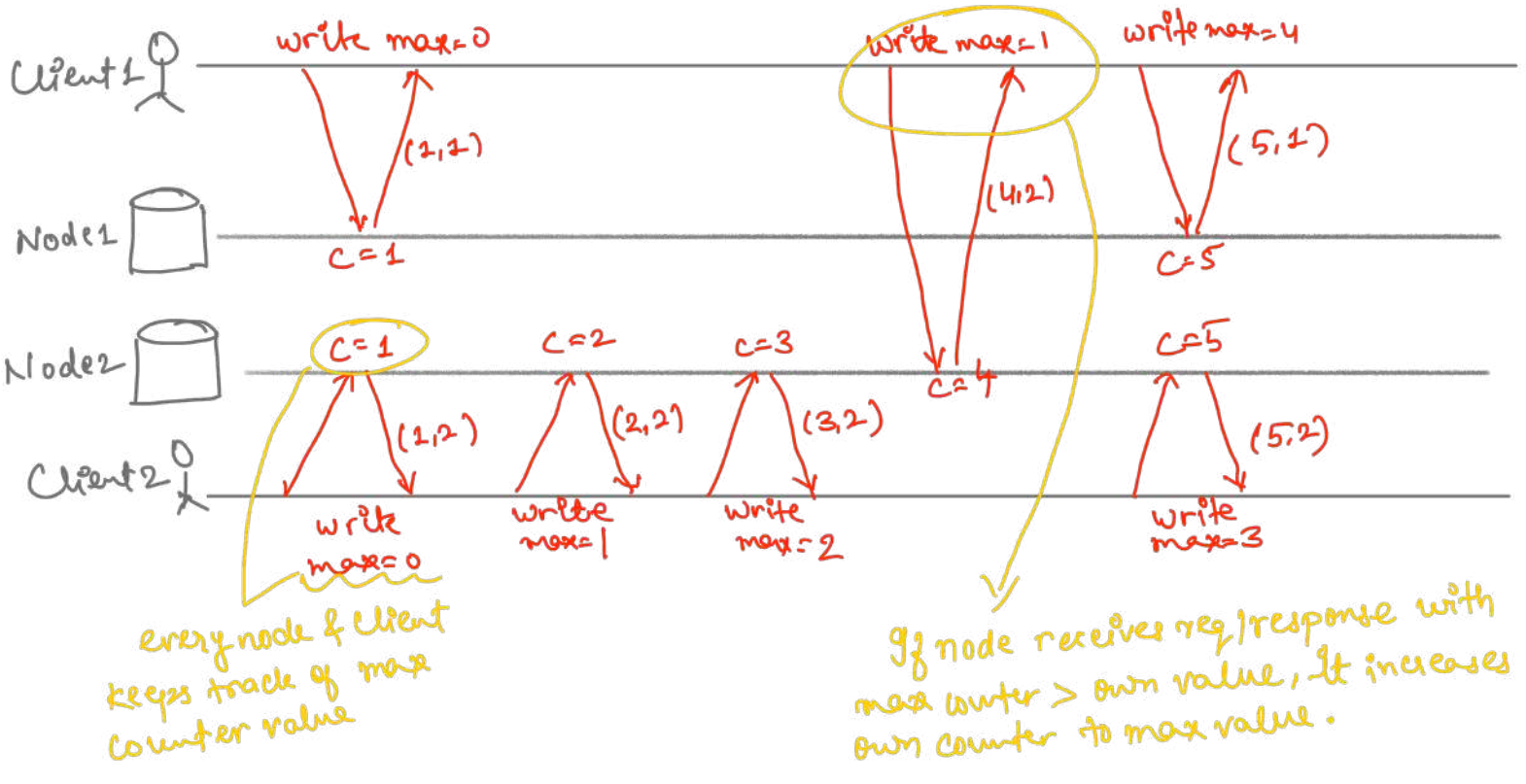
example multi leader replication

1. Each node generate its own numbers.
   node should have some identifier so as numbers are unique across clusters
   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

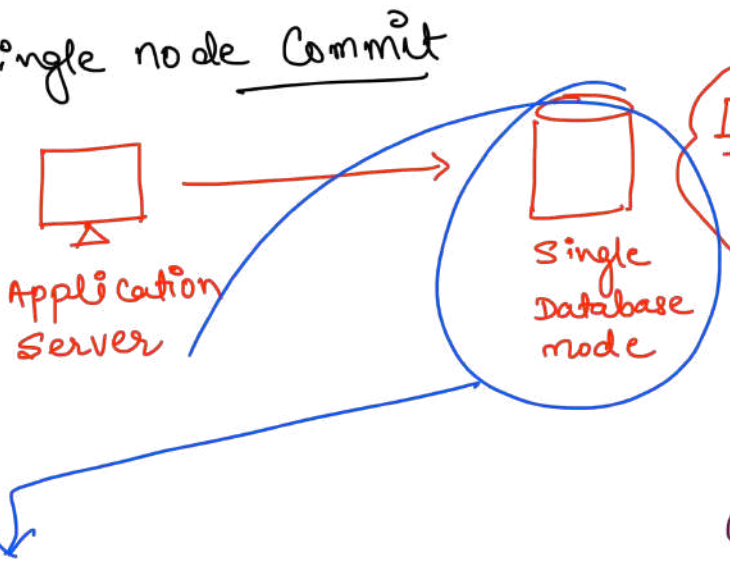
↳ If msg is already delivered, node is not allowed to insert it back in earlier position.

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

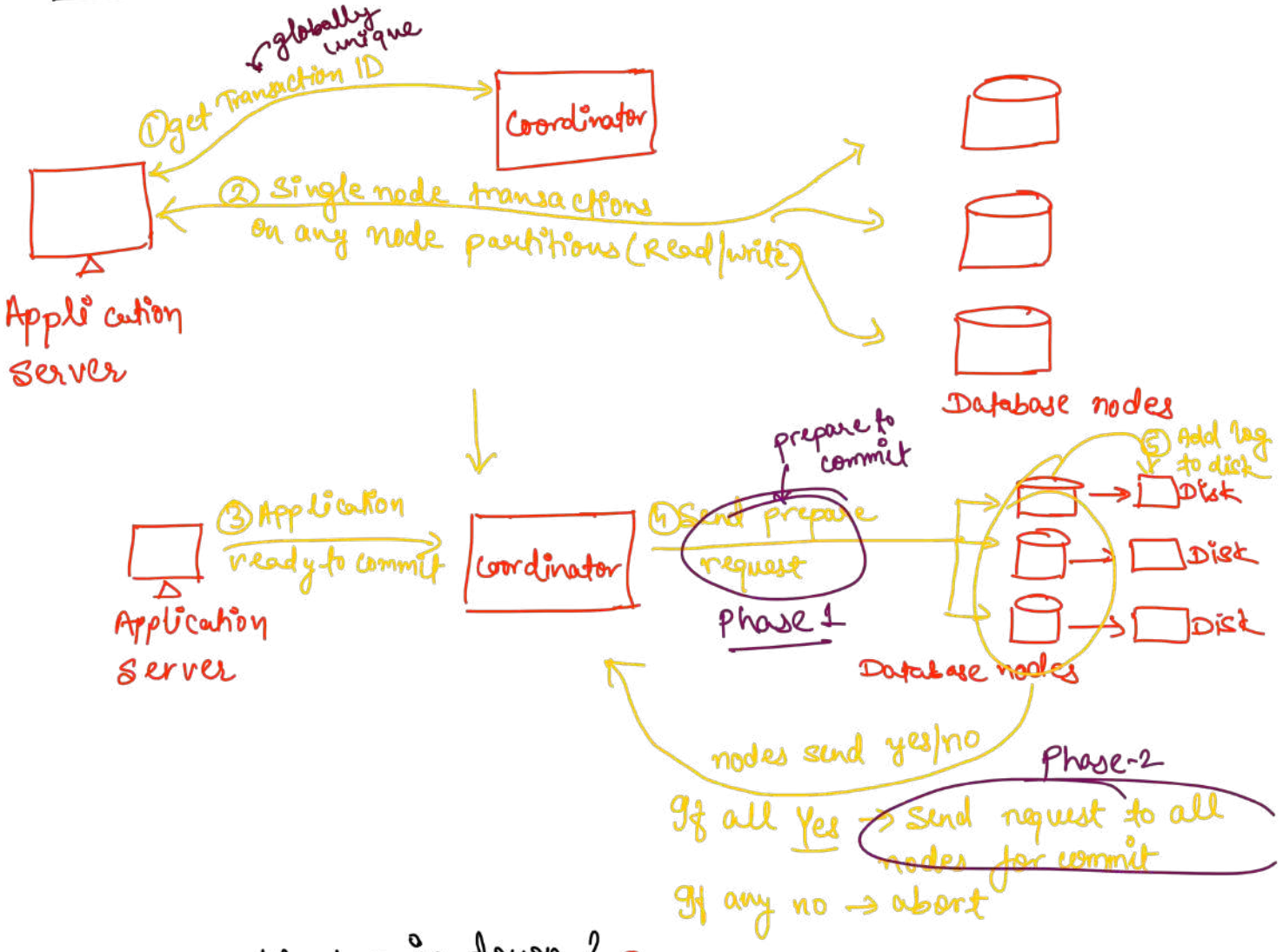
What if crashed after #2?
↳ recover from disk once node restarts.

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?

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.

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

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

extentended 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... ~