CLOUD COMPUTING

UNIT-3

Distributed Storage

feedback/corrections: vibha@pesu.pes.edu

VIBHA MASTI

Storage

1. File

- Hierarchy of files, folders
- Access via path
- Limited metadata

2. Block

- Chunks of evenly sized volumes
- Each block: unique ID
- Underlying storage software reassembles data from blocks when requested
- Large amounts of data
- Low metadata
- Expensive

3. Object

- Flat structure
- Metadata important (2 levels)
- Unique ID

Cloud Storage Architecture

1. User access layer
2. Data service layer
3. Data management
4. Data storage

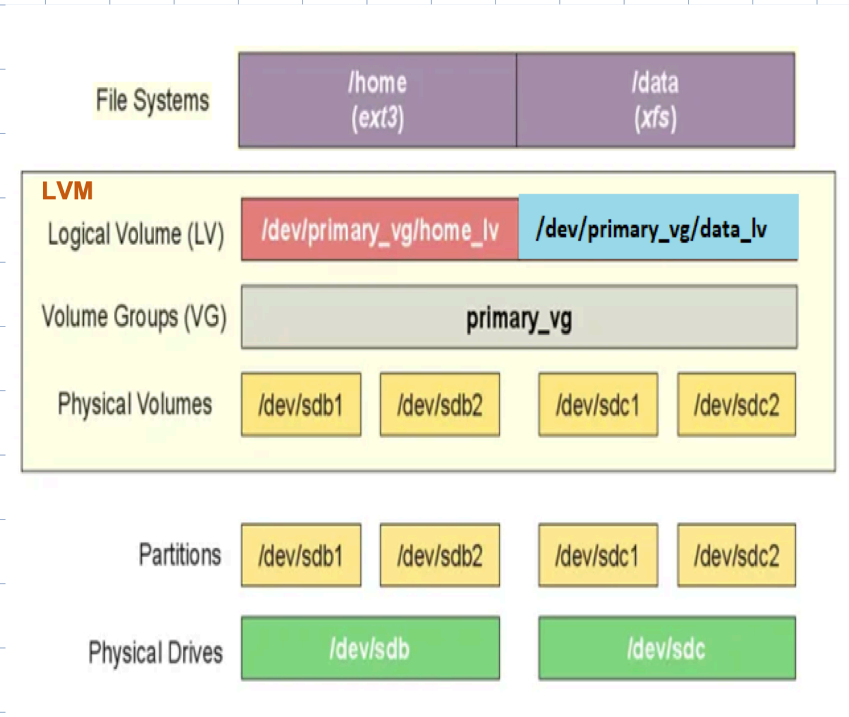
Cloud Storage Virtualization Enablers

(i) File Systems

- Data grouped into files
- Network FS

(ii) Logical Volume Manager (LVM)

- Layer b/w FS and disk drives



(iii) Thin / Virtual Provisioning

- Storage perceived by app > physically allocated storage

Categories of Storage Virtualization

1. File-Level Virtualization

- Abstraction of FS to app
- Single logical FS over distributed FS
- Distributed FS - managing metadata

* Centralized metadata server

↳ dedicated server for MD
↳ lock-based sync
↳ bottleneck
↳ scale well for large files
↳ Lustre

Lustre

- 3 components - OSSes that store files on OSTs, single MDT that stores MD on MDSes, Lustre clients
- Read functioning

* Distributed data & metadata

↳ greater complexity
↳ Gluster FS

Gluster FS

- 2 components - client & server
- Server "clusters" all physical storage servers and exports combined diskspace as GFS
- Storage brick
- Translator

2. Block Virtualization

- Single logical disk
- 3 levels

* Host-Based

↳ LVM

* Storage Device Level

↳ virtual volumes over physical storage
↳ RAID
↳ Logical Units (LUNs)

* Network Based

↳ Most common
↳ Within network connecting hosts & storage
↳ switch or appliance based

in-band   out-of-band

3. Object Virtualization

- Flat namespace
- File + system MD + custom MD
- Use REST APIs
- Critical tasks

* data placement
* automating management tasks

- Durability

* replication
* erasure coding

Amazon S3

- Buckets contain objects
- Replicated in many geo locs (objects)
- Keys: <opt-dir-path>/<obj-name>
- Security
  * Access control to objs
  * Audit logs
- Data protection
  * Replication
    - ↳ survive 2 replica failures
    - ↳ RRS survives 1 replica failure
    - ↳ consistency
  * Regions
    - ↳ geo area
    - ↳ legal, availability reasons
    - ↳ bucket region
  * Versioning
    - ↳ full history
- Large objects - multi part uploads

OpenStack Swift

- Swift partitions: locations for data from available storage
- Account: user in storage system
- Containers: where accounts created, stored (namespaces)
- Object
- Ring: maps partition to physical locations

DynamoDB

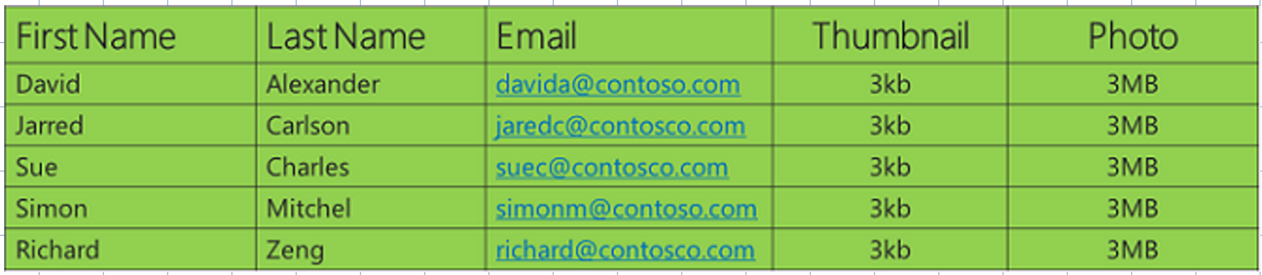
- NoSQL, key-value, AWS
- Tables created in advance
- Primary key, secondary index
- Item-level consistency
- No joins
- Primary keys
  - * Partition key
  - * Partition key and sort key

PARTITIONING

- Partition datastores
- Skewed, hot spot
- 4 approaches

(i) Vertical Partitioning

- column
- No 2 critical columns together



First Name	Last Name	Email	Thumbnail	Photo
David	Alexander	davida@contoso.com	3kb	3MB
Jarred	Carlson	jaredc@contosco.com	3kb	3MB
Sue	Charles	suec@contosco.com	3kb	3MB
Simon	Mitchel	simonm@contoso.com	3kb	3MB
Richard	Zeng	richard@contosco.com	3kb	3MB

(ii) Workload-driven

- Data access patterns

(iii) Random Assignment

- Disadvantage: query all nodes in parallel

(iv) Horizontal

- Static
- 4 techniques

1. Key Range

- Disadvantage: data access can lead to hotspots

2. Schema-Based

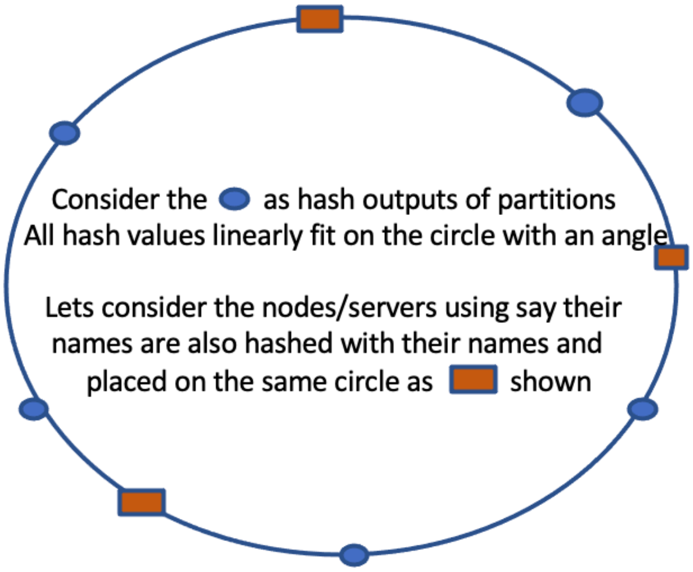
- Related rows together

3. Graph

- Workload based
- No repartitioning

3. Hash

- Evenly distributes rows
- Cannot do efficient range queries
  ↳ concatenated index
- Distributed hashing
- Rehashing
- Consistent hashing : independent of # servers



Consider the ● as hash outputs of partitions
All hash values linearly fit on the circle with an angle

Lets consider the nodes/servers using say their names are also hashed with their names and placed on the same circle as 🟧 shown

- Secondary index
  - * Document-based partitioning (local index)
  - * Term-based partitioning (global index)

Rebalancing

1. hash mod N
   - Range of hashes to a server
   - Expensive to rehash

2. Fixed no. of partitions
   - Assignment of part to node changes (size of part changes)

3. Dynamic
   - key range partitioning
   - Split & merge partitions dynamically

4. Partition Proportionally to Nodes
   - Fixed no. of parts per node
   - No. of parts $\propto$ nodes
   - Hash

Request Routing

- Service discovery problem
- Approach #1
  - * clients contact any node (RR load balancer)
  - * if part not present, forwards to appropriate node
- Approach #2
  - * all reqs to routing tier
  - * partition-aware load balancer

- Approach #3
  * clients aware of parts

- Approach #4
  * Zookeeper
  * nodes register in ZK
  * routing tier or client can subscribe to ZK
  * ZK notifies RT of updates

Replication

- Writes need to be processed by every replica
- Algorithms
  * single-leader
    → Sync
    → Async
  * Multi-Leader
  * Leaderless

(i) Single leader

- One replica is leader
- All writes to leader
- Leader sends replication log to followers
- Followers perform writes in order
- Followers: only read queries

* Synchronous
  ↳ Leader waits for follower to confirm before reporting success
* Asynchronous
  ↳ No wait

Implementation of Replication Logs

1. Statement-based
2. Write-Ahead Logs
3. Change Data capture | Logical Log
4. Trigger based

Potential Issues

(a) Follower failure

- Catch up recovery - logs
- Follower requests data changes from leader (since failure)

(b) Leader failure

- Failover
- Choose new leader and reconfigure clients
- Manual or automatic

(c) Replication lag

- App may read outdated info from async followers
- Apparent inconsistencies — eventual consistency
- Delay between write to leader and replication on follower
- Identification: read your own writes
- Possible solutions
  - * Read critical data from leader
  - * Monitor lag & prevent queries on follower with large lag
  - * Client last write timestamp
  - * Monotonic reads
  - * Consistent prefix reads

(ii) Multi-leader

- Useful for
  - * Multi datacenter operation
  - * Clients with offline operation
  - * Collaborative editing
- Write conflicts
  - * Conflict avoidance: writes for particular record through same leader
  - * Converging towards consistent state : each write unique ID
  - * Custom conflict resolution

(iii) Leaderless

- All nodes accept reads & writes
- Write success if ACKed by quorum of k replicas
- Read success if quorum of k reads agree on a value

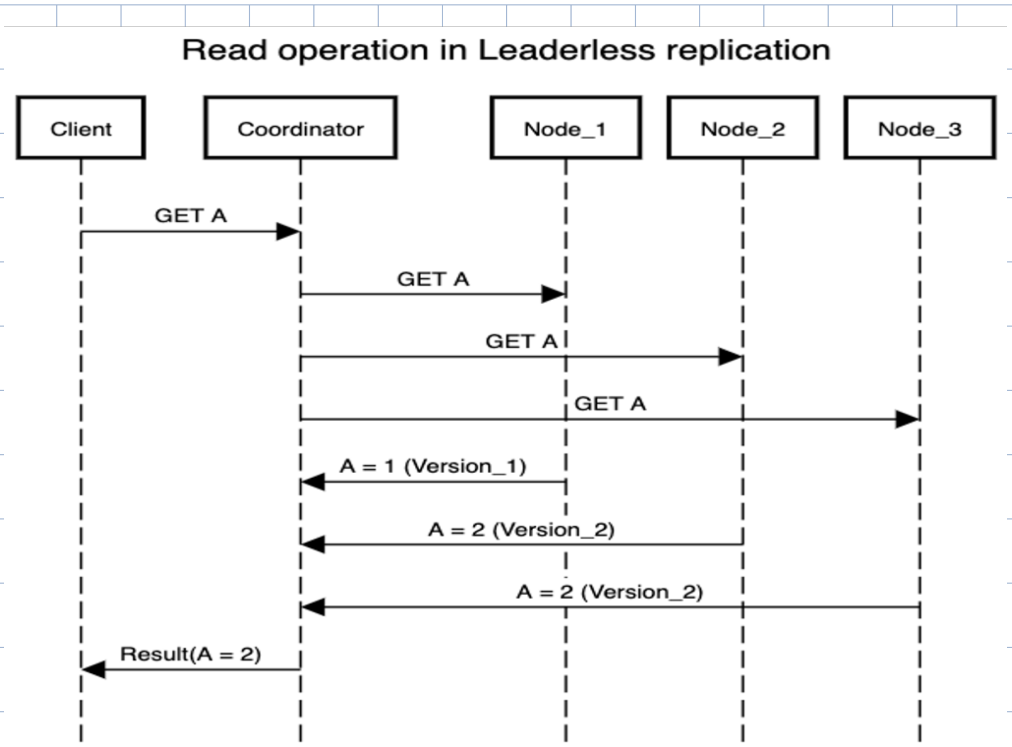
* Write

- client broadcasts req to all replicas
- waits for certain no. of ACKs
- success if ACKed by k out of n replicas

* Read

- client contacts all replicas
- success if k out of n reads agree on a value

- k for read & write may differ



- Read repair: client sees outdated value in replica and updates with new value

- Concurrent writes:
  * Last write wins
  * Version numbers

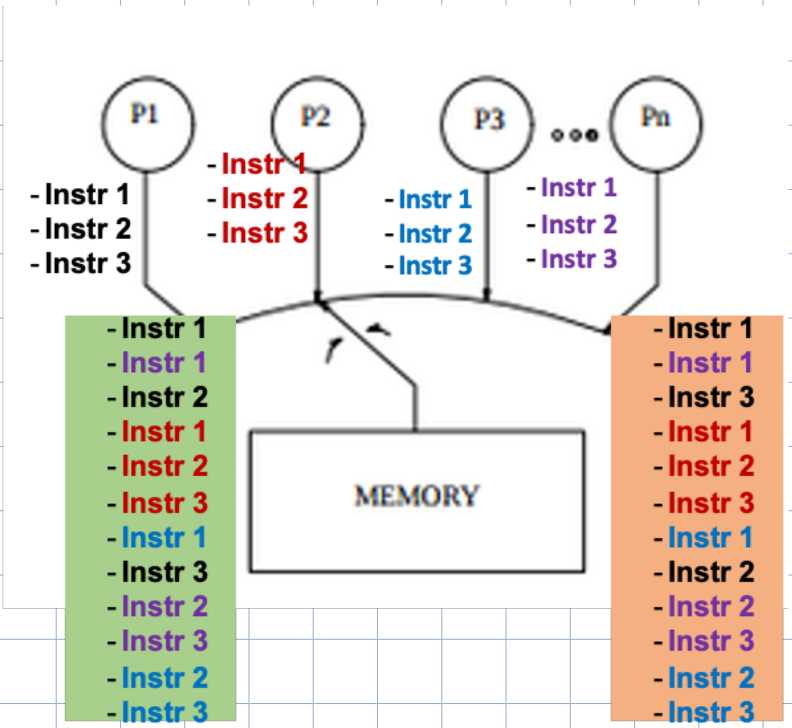
CONSISTENCY MODELS

1. Eventual consistency

- Stop writing, wait for unspecified length of time, eventually all read reqs return same value
- Weak guarantee

2. Sequential consistency

- All processes see same order of all memory access operations



3. Causal consistency

- 2 events causally related if one can influence the other
- weaker than sequential
- all processes observe causally-related operations in a common order
- If sequentially consitent, also causally

4. Pipelined RAM (PRAM) consistency / FIFO

- All writes performed by single process seen by all other processes in the order in which they were performed

5. Strict / Strong Consistency / Linearizability

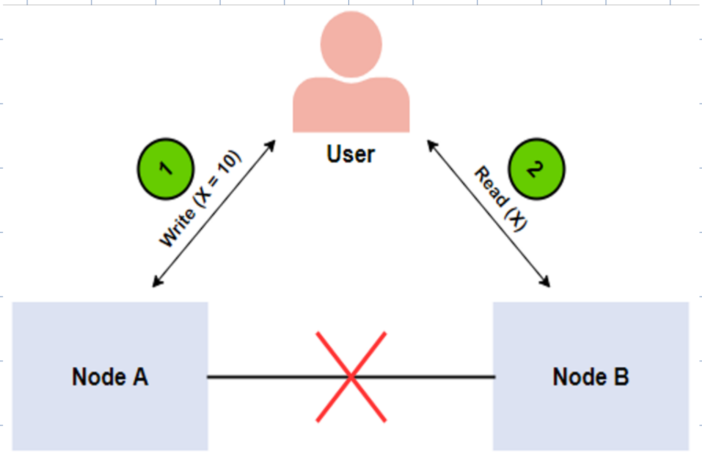
- Appear as a single-copy system
- Atomic consistency
- Read guaranteed to see most recent write
- Check timings of all reqs and res and check if they can be arranged into a valid sequential order

CAP Theorem

- Consistency (same copies of item on all nodes)
- Availability (all working nodes return valid response)
- Partition tolerance (re-route on fail)

- Cannot have all 3 in a distributed system with data replication

- Eg: DS of 2 nodes storing value of X. Network partition happens. Either C (no A) or A (no C)

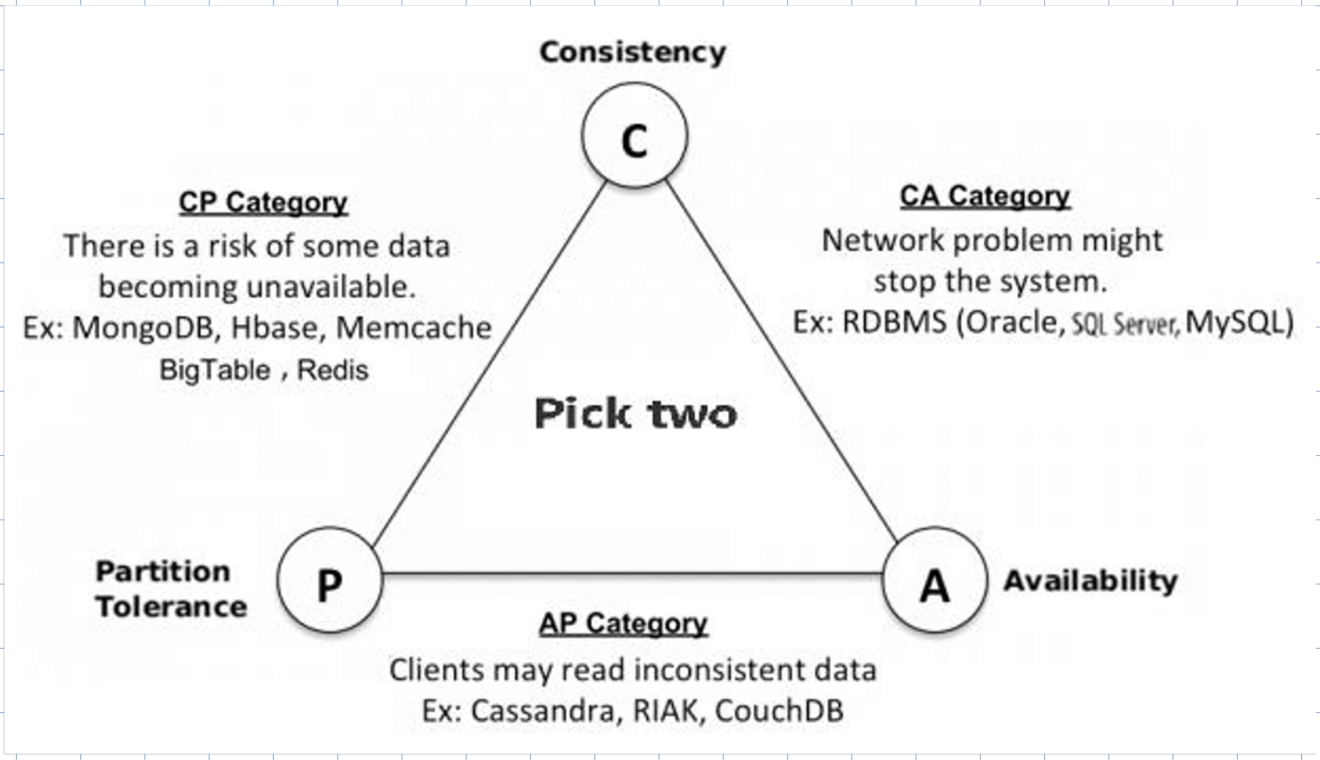


Tradeoffs

1. ***Availability and Partition-Tolerant*** ***(Compromised Consistency)***: Say you have two nodes and the link between the two is severed. Since both nodes are up, you can design the system to accept requests on each of the nodes, which will make the system available despite the network being partitioned. However, each node will issue its own results, so by providing high availability and partition tolerance you'll compromise *consistency*.

2. ***Consistent and Partition-Tolerant*** ***(Compromised Availability)***: Say you have three nodes and one node loses its link with the other two. You can create a rule that, a result will be returned only when a majority of the nodes agree. In-spite of having a partition, the system will return a consistent result, but since the separated node won't be able to reach consensus, it won't be *available* even though it's up.

3. ***Consistent and Available*** ***(Compromised on a Partition-Tolerance)***: Although, a system can be both consistent and available, but it may have to block on a *partition*.


Consistency
C
CP Category
There is a risk of some data becoming unavailable.
Ex: MongoDB, Hbase, Memcache BigTable , Redis
CA Category
Network problem might stop the system.
Ex: RDBMS (Oracle, SQL Server, MySQL)
Pick two
Partition Tolerance
P
A
Availability
AP Category
Clients may read inconsistent data
Ex: Cassandra, RIAK, CouchDB

DISTRIBUTED TRANSACTIONS

- Begin
- End (try to commit)
- Abort (kill)

- ACID
  ↳ Atomic
  ↳ Consistent
  ↳ Isolated | Serializable
  ↳ Durable

Transactions

1. Nested transactions

- Sub-trans may commit, parent may fail
- Solution: private workspace
- Commit: pvt copy displaces parent's

2. Distributed transactions

- Each node has
  - local transaction manager
  - transaction coordinator (nominated from LTMs)

Commit Protocols

- Ensure atomicity
- Commit at all sites or abort at all sites

Two-Phase Commit

1. Phase 1

- Coordinator writes 'prepare T' on its logs
- Coordinator sends 'prepare T' to all sites

- If site ready to commit, enters pre-committed state
- Site places 'ready T' on its logs
- Site sends 'ready T' to local coordinator
- Once site in pre-committed state, cannot abort unless told to by coord

- If site not ready to commit, writes 'don't commit T' to local log
- Sends 'don't commit T' to coord

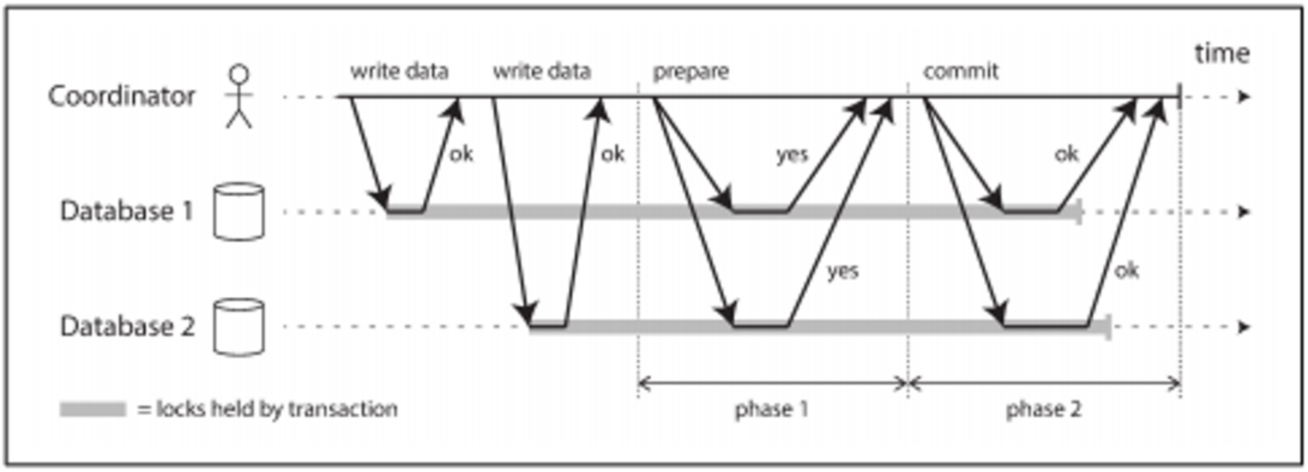
- Coord waits for all nodes to reply

2. Phase 2

- If all nodes say 'ready T', decides to commit
- Coord logs 'commit T' and sends

- If one or more say 'don't commit T', logs 'abort T' and sends

- Site either commits and releases locks, logging 'commit T', or aborts and releases locks, logging 'abort T' (based on received message)


time
Coordinator
Database 1
Database 2
write data
ok
write data
ok
prepare
yes
yes
commit
ok
ok
phase 1
phase 2
= locks held by transaction