→ Difference b/w distributed & parallel systems

↓

Single consistent view
(Synchronization & Retrieval)

Course Content:

- RPC → Networking
- Logical Clocks
- Distributed Deadlocks
- Paxos / Raft → Papers
- Google big tables / Hbase / NoSQL
- Amazon Dynamo

Assignments:

1) Serialization / Deserialization
2) Grossip Protocol
3) HDFS + Map Reduce

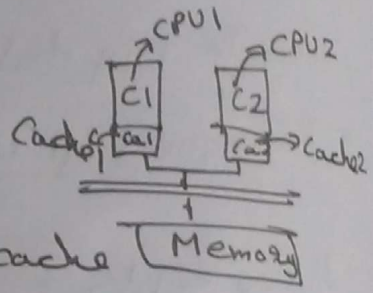
---

* Flynn's Taxonomy: (Assembly Operations)

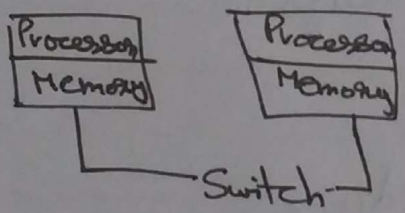
- SISD → Single Instruction single data
- SIMD → Single Instruction multiple data
- MISD
- MIMD → Parallel & Distributed Nature
  - → Shared Memory Systems → Multiprocessors
  - → No Shared Memory → Multicompactors (Networked using switches)

⇒ In Shared Memory Systems

- They are not scalable
- Cache coherence problem.
  ↳ Each CPU has its own cache



⇒ In No Shared Memory



Note: bitfields ⇒ int F1:2 , F2:1;
Allocate 2 bits to F1 & 1bit to F2
(Space optimization)

---

* When using bitfields on threads, we need to use locks. → Cache coherence issues also arise.

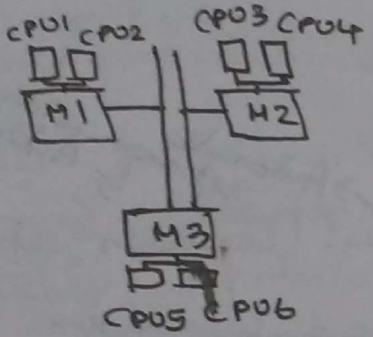
Note: * Write through cache ↓ Always update memory & Write back cache ↓ Update memory on eviction.

Cache Coherence:

- A hardware solution where every cache keep checking their bus for updates on cached data. Not very effective & might fail. (Snoopy cache etc)
- Software solutions → Locks on critical sections.

———x———

* NUMA Systems → Hierarchical memory (Shared Memory System)
(Non uniform memory access)



- CPU1 can access memory M1 quickly but M2 & M3 take more time.

- The only benifit is NUMA reduces contention (load) on the BUS.
(Memory contention)

———x———

Networked File System: (NFS)

- Storage done on a single place → NFS Server

———x———

Note: Big Endian vs Little Endian

Note: NTP → Network Time Protocol → Clock synchronization b/w various systems

Note: MapReduce programming.

———x———

Communication:

- Different frequencies → Sharing a medium
- Circuit Switching & Packet Switching

Circuit Switching	Packet Switching
→ Guaranteed fixed bandwidth	→ Packets
→ Fast, only circuit ID needed.	→ No resource loss etc
→ TDM (Time division multiplexing)	→ Out of order delivery.

—— x ——

Note: Why does TCP require handshaking? (Connection establishment) → Automatic retransmission & Acks.

—— x ——

Note: * ASCII vs EBCIDIC, Marshalling & Unmarshalling, Endian → Similar to serialization

—— x ——

Note: * htons → host to network short
htonl → host to network long
ntohs } Reverse
ntohl }
(Idempotent functions) —— x ——

→ Convert to a format easy to store & transfer

* 7-layer network stack
→ Physical, DataLink, Network, Transport, Session, Presentation & Application.
→ SSL & Encryption

—— x ——

Note: resolv.conf ⇒ Set addresses of primary & secondary DNS servers. (If not using DHCP)

/etc/hosts ⇒ Map different addresses to names as we want to.

—— x ——

(CS 417 Rutgers University)

Note: • Non blocking socket I/O
• select, epoll system calls
• Read Java's RMI Server implementation.
• Why do we need non-blocking socket I/O.
↳ If client sends 5 bytes & server is waiting for 10 bytes, we have essentially wasted a thread.

4

(Install google protobuf)
↳ Look at serialization code.

*Remote Procedure Calls: (RPCs)

- Client & Server stubs take care of serialization (Proxy) (Skeleton) etc & all other network routines.

⇒ We just need to write client & server functions

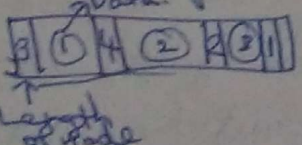
a) • The client stub should ensure that the client feels its ~~nothing~~ executing functions directly instead of the client-server thing, where the server actually runs the code & sends response

b) • The server stub would take care of handling all the requests & calling the required server functions etc.

- RPC essentially hides network details such as sockets, encodings etc.

I) ↳ Parameter passing: (Serialization / Deserialization)
- Pass by value, reference, data structures
- Try to avoid copying

↳ For linked lists, we could put data in a string kind of structure where,



II) ↳ Representing Data

- Network formats like XML, JSON, Google Protobufs, XDRs &c.

III) ↳ Locating Servers

* • Portmap server (portmapper) runs on a certain port (well known)

- All services register themselves with the portmapper and each host can maintain locally provided services.

IV) ↳ Error Handling

- These are 4 of the things RPC system has to implement.

—— x ——

Note: An RPC call may be executed 0, 1 or more times for various kinds of error handling.

—— x ——

~~RPC Functions~~

* • RPC Semantics

↳ Idempotent functions
  - Can be run multiple times without harm

↳ Non idempotent functions
  - Should be executed only once successfully

—— x ——

* Programming with RPCs: (RPC Compilers)

- IDL — Interface Definition Language
- We define RPC in IDL file, then IDL compiler generates everything needed.

Note: ONC (Sun) RPC is very famous, mainly because of its NFS. — rpcgen

———×———

*Java RMI:

- Simplifies the RPC implementation in Java

———×———

5

Note: Name Server ⇒ Helps find services by giving pointers
* (Present on server machine) a pointer to the service, so we can invoke functions on pointers etc

- Server implements a "Remote" & "Lock" Interface (Serializable interface could also be there)

———×———
———×———

*Clocks:

- Physical clock synchronization
  ↳ Quartz oscillators dont oscillate exactly, so synch is hard.
* ↳ Clock Drift ⇒ Clock tick rate change
  ↳ Clock Skew ⇒ Difference between 2 clocks.
- A battery keeps the clock on even when powered down (In a system).

⇒ We could use a time server with request/response protocol. Transmission delays might be an issue.

I) Cristian's Algorithm: → Server sends
* • Send request at t_1 & get response at t_2, we assume time stamp actual is at $t + \frac{(t_2 - t_1)}{2}$. → We set time on client to this. (Assumes server time is correct)
  • Assuming network delays are symmetric.

II) Berkeley Algorithm:

- Get average from all participating systems.

⇒ Update all clocks to average.

- So we don't assume server clock is always correct.

Note: NTP – Network Time Protocol (Nowadays what is used for sync), PTP

b

*Logical Clocks: (Try maintain total ordering)

- A simple counter could be used where every event increases the counter by 1.
- $a \rightarrow b \Rightarrow clock(a) < clock(b)$
  - a, b are causal events
  - a caused b to happen.

⇒ For any 2 events either,

$a \rightarrow b$ (Causal) or $b \rightarrow a$ (Causal) or ($a \not\rightarrow b$ & $b \not\rightarrow a$) (a, b are concurrent events.)

*Lamport's Algorithm:

- When a message arrives

  if receiver's clock < message counter timestamp

  ⇒ Set receiver's clock = message counter + 1.

- This would ensure that with multiple systems

$a \rightarrow b \Rightarrow clock(a) < clock(b)$

(If receiver's clock > message, we can choose to ignore it (the message))

- We would need to store the clock counter on disk, but every write is expensive to disk.
  - So every now and then if our counter is x, we store x+200 on disk & then we don't need to write to disk again till we reach x+200, and then we write again. (Amortizing the write cost)

(On reboot in case of power cuts etc, we start with the written clock counter again, so we are never going back in time).

⇒ This algo is good but we want to have,

$C(e) < C(e') \Rightarrow e \to e'$ (For causal events)

(This is the opposite of what lamport's algo provides) ($e \to e' \Rightarrow C(e) < C(e')$)

↳ Solution = Use vector clock.

---

* Unique Timestamps:

- In lamport's, we might face an issue that clock(a) = clock(b), we don't want this.

⇒ So we give every process a unique ID & the timestamp = (clock, ID)

- So $(T_i, i) < (T_j, j)$ iff $T_i < T_j$ or ($T_i = T_j$ & $i < j$)

(This might not capture the original occurance of the events but its fine.)

---

* Vector Clocks: (Slides for example)

- A vector is maintained. Each process i updates V[i] with its current timestamp.
- Process i on sending does V[i] += 1, and process j on receiving does the same.
- If we receive a message, it will update each element of its vector with max of itself or message vector.

- A problem that occurs here is we dont know the no. of processes initially.

⇒ If at least one element of V is lesser for one of the events and rest are lesser or equal, then we can assume that if $V_a \leq V_b \Rightarrow a \rightarrow b$ (Causal)
with 1 or more elements strictly lesser

---

* Generalized Vector Clocks:

- Similar to vector clocks, but we store tuples

{"identity1": value1, "identity2": value2, ... }

- In normal vector clocks, we use indices of the vector for identity of processes, but here we use the keys of the dictionary.
- This would take care of processes being added/removed etc.

Note: Protobuf makes keys as integers
- → So it becomes smaller representation
- → Faster to encode/decode etc.

---

* Totally Ordered Multicast: (Very inefficient) (Very slow)

- Using logical clocks,

⇒ Each message is sent to all nodes by sender (O(n))
- A local queue stores message list.

⇒ Ack is sent from all participants to all (O(n²)) participants → So their clocks get updated to later than sender
- Message received if:
  - Message at top of queue
  - All acks received.

} Now the sender can perform the required write etc

Since all other clocks

Note: Just TCP won't ensure message ordering. If multithreaded systems etc, then we need to take care.

—— x ——

Note: Synchronous, (Synchronized)

Handling failures:

- If a server goes down & then back up then it is fine to serve read requests with stale data but it won't accept write requests till it has synchronized with other servers.
- If all servers go down, on booting back up, compare clocks to see who applied most updates and everyone else copies the differences.

⇒ Store clocks → Locally
- Single point of failure, so = servers, might just vanish completely

→ Centralized storage.
- We know that if a server goes down, then its clock doesn't update etc. (So we know that there is a problem)
→ (Need to make this highly accessible)

* ——— x ———

8

* Distributed Shared States & Mutual Exclusion:

* 1) Centralized:

- Single point of failure
- A centralized lock server which gives access to clients to take ownership of a lock etc.

⇒ Lock server would store,

Resource → Owner, List of waiters.
R1 C1 C2 → C3 → C4

- Some issues may come up, deadlock conditions,

*- Some of the issues are,

↳ Lock holder dies

↳ Solutions could be, lock server periodically polls and withdraws lock if holder is down.

↳ Issue a timeout, so if holder is down, time is not renewed.

↳ Multithreaded applications

↳ Use the pid & tid for identification

↳ Dead, lock server → Crashed

↳ We could persist the request queue, but other problems like atomicity, storage on disk etc come up.

↳ We could make the clients aware that the server went down, so they send lock requests again

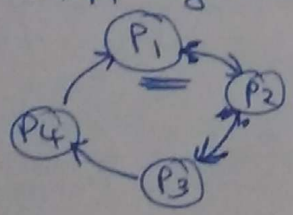
↳ Either each client heartbeats at regular intervals

↳ Or once server is up, it sends a message.

*2) Token Ring:

- Assume a logical ring can be formed among the processes.

⇒ A logical token is passed around.



- Hold token while you are accessing the critical section.

- Issues:

↳ If a token is lost, how can this be detected.

↳ Recreation of token ensuring there is only one token always.

*3) Lamport's Algorithm: (Slides for details) (Video) (TOM for locks & mutual exclusion)

- Each process has a request queue of mutex requests

⇒ Messages are timestamped with totally ordered Lamport timestamps. (3(N-1) per request)

- Request sent as request(i, Ti) sent by Pi
  - → Add req to own queue.
  - → All other processes ack and put request on their queue in correct timestamp ordering.
  - → After receiving all acks, Pi enters critical section and req is in front of queue.
  - → Release critical section with release(i, Ti), all the processes remove req from their queues

*4) Ricart & Agrawala's Algorithm: (Assign 2)

(If a release is received, also treat it as an ack)

- Similar to Lamport's but with small changes

⇒ Ack & Release messages are combined so that the overall number of messages are reduced. (2(N-1) to 3(N-1) per request)

- If a process knows its req is top on queue, then it won't send ack to another req since this will use the critical section.

———x———

*Elections:

1) Bully Algorithm:

*
- Select process with largest ID as coordinator

⇒ Each process sends a message to all processes with larger ID. If anyone responds, we are done.

⇒ On getting a request, we send back message, accept, and then repeat the process.

- If no larger process ID present (ie. no response) then current process becomes coordinator.

⇒ New coordinator announces victory.

- Issues
  - ↳ We send accept & then crash. So election would freeze. We need to keep timeouts etc.
  - ↳ Dead process recovers, election is held to find the coordinator.

2) Ring Algorithm:

- Logical ring arrangement
- On detecting a process failure, create election message with pid. If successor is down, hop over it.
- On receiving an election message, add pid to body and pass it on.

⇒ the process who started the election (sees its pid in list) decides on a leader and then lets everyone else know in a similar circular manner.

- Issues
  - ↳ Multiple election messages
  - ↳ Process dies after adding itself, can cause lots of issues.

3) Chang & Roberts ring: (Still few issues exist) (Slides)

- Similar to above, but ensure only one election is going on at a certain time.

→ If we get an election message but already a part of election, drop the message (If message is lower than my id priority)

- Don't store list of pids but store lets say largest / smallest pid & pass along.

Note: • route
linux
• Zookeeper
• btree

* Split Brain (Partitioning)

- Network partitioning happens.
- Multiple nodes may decide they are the leader.
  - → To solve this insist on a majority.
  - → Alternate communication to validate failure.

Note: Failure Types: (Slides for full list)

*
- Software failure
- Node crash
- Hardware issues
- Node restart
- Hung kernel, Power failure etc
- Network issues

⇒ RPC caller detects timeout & hence failure maybe due to infinite loop in server or slow server etc.

10

---

* Transactions: (Not for exams)

1) File copy/creation:

- Add new file entry to btree.
- Create inode corresponding to new file
- Allocate space & mark bit map as used.
- (Optional – Split btree if no. of pointers are too many)

---

* ⇒ Write Ahead Logs: (WAL)

*
- Put all the things that need to be done for a transaction of the WAL. This is fast since these writes are small.
- WAL is generally a contiguous block on disk.
- All the actual updates are done in block wise memory & once a ~~transaction~~ block update is complete it updates on disk.

- Suppose the system crashes when applying updates on memory. On reboot, all the memory is reset, so we go through WAL and then perform all the unfinished transactions from start again. Once memory has been updated, we move it to disk again → just like before

WAL ⇒ Stores (i, j) where i = Task, j = Transaction id

ex: WAL ⇒ [(1,1), (2,1), (1,2), (1,3), (2,2), (3,2), (2,3)]

- All tasks of a transaction need not be next to each other. (Assume task 2 = End transaction)
- If the "End transaction" message is not marked ~~here~~ as complete on WAL, then that entire transaction is repeated.

—END of Mid 1—

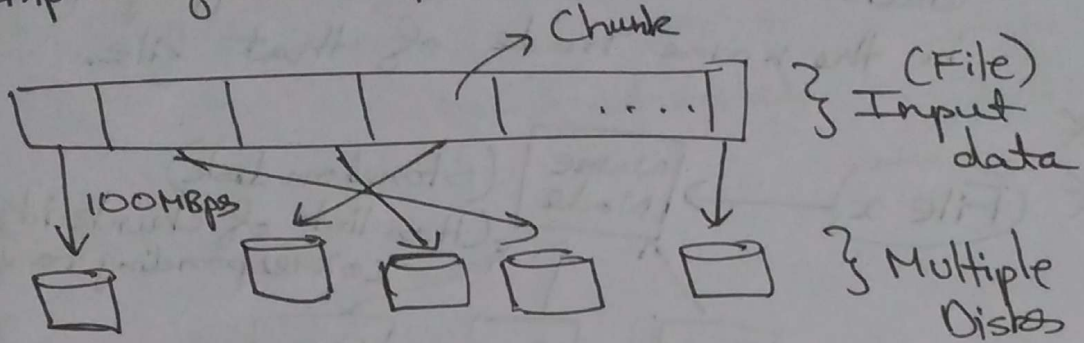
Note: Assignment 2

- Queue modifications → Require local locks?
  - RMI does request handling synchronously or asynchronously?
  - → Since access same resource (queue) on multiple threads?

HDFS (Hadoop)

(Inspired by Google File System) (GFS)

* • Improving write speeds.



∴ With lots of disks we get k×100MBps with k disks.

↳ Horizontal Scaling paradigm.

⟹ For disk failure issues etc, use backup disks or RAID.

* • How do we read these chunks back & get back original data?

↳ For each chunk we need to know which chunk id is stored on which disk(s) → This is if we store each chunk on multiple disks to address failure.

↳ We also need to maintain [redacted] which our next chunk id is. (They need not be in order)

FS image on NN.

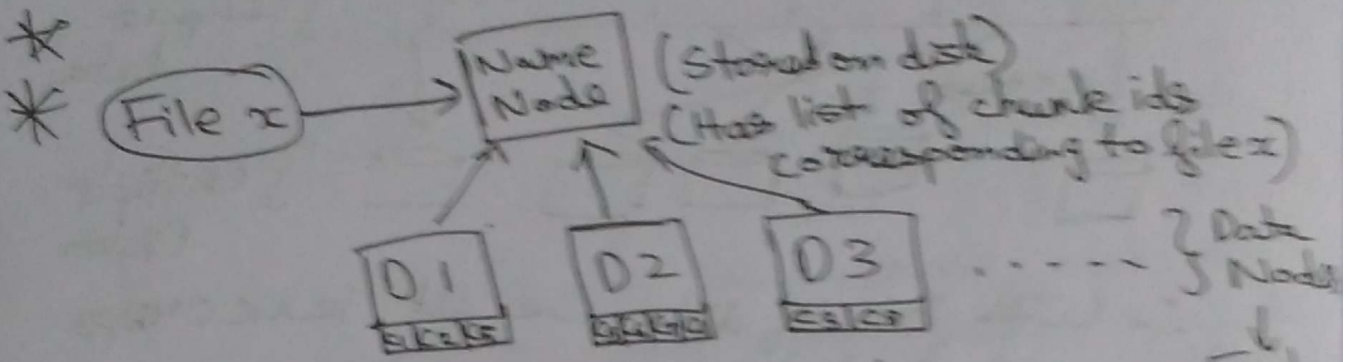
ChunkId	DiskNum(ip)
C_1	D_1, D_3, D_5
C_3	....
C_4	
C_5	
C_{10}	

For Each large file.

* • Ideally store everything on disks so we have data on [redacted] restart.

• But we can do it by just storing the chunk ids in order
↳ Name Node stores this.

⇒ What happens is that, the chunks stored in data nodes are known to that data node. So the data nodes periodically making the name node letting it know which chunks it has, so the mapping gets restored in the name node of that file.



---

* Note: HDFS assumption → Write Once Read Many (So they exploited this through)
  ↳ Use 128MB chunks

---

* Writes in HDFS:

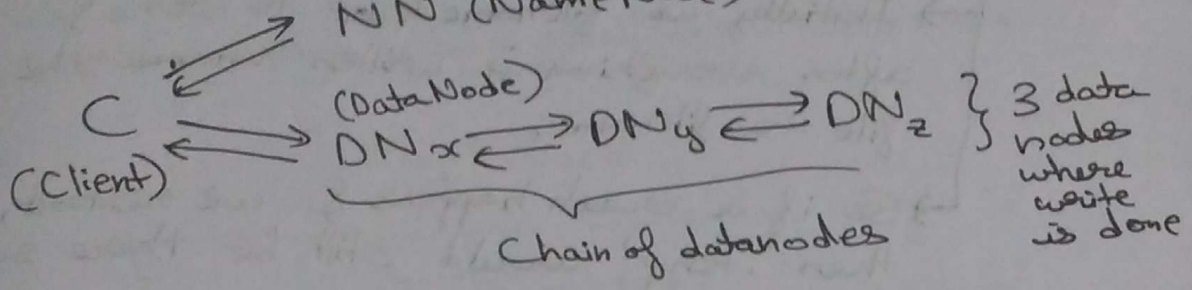
Create a/b (File)
write() ⟶ Give me a chunk id → #1000 (Name Node: a → b)
← D1, D5, D7

- Write 128MB chunk with given id to the 3 given disks.
- Also store the chunk ordering

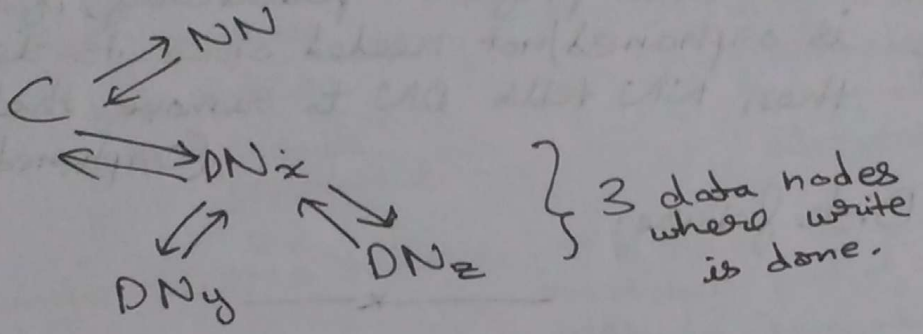
- The name node does this by checking disk utilization, bandwidth and if they are on different racks etc

---

1) Cascaded Chain: (HDFS supports this)



2) Star:



- Cascaded is better if we consider throughput.
  - ↳ In star, the x node needs to split throughput & do y & z, so it will take longer to complete write. (Can we send both parallelly?)

—— x ——

* Failures:

1) NN dies → While client has written on let's say 2 DNs, but now doesn't know which is the 3rd DN. So it needs to clean up the data (Transactions)

2) Client Dies:
   - → DNs that are still on pending transactions need to be removed
   - → open handle in NN tree. (Needs to be removed)

3) DN dies:

↳ Initially we filename.temp to store & after everything is written rename the file to filename.

↳ So if a crash happens & we restart, then '.temp' would still be there so we retry.

Note: When DNs ping NN periodically, if a chunk is orphaned/not needed since its deleted, then NN tells DN to remove that chunk. (Orphaned chunk)

4) Disk failure:

———×———

Note: We cannot reuse chunk numbers since on reuse, if we ask to write on 3 DNs where that chunk already exists then on a DN failure & coming back up, it would still have same chunk number but old data. So this is an issue. So we don't reuse chunk numbers.

———×———

Note: Name node → Single point of failure (HDFS Issue)
↳ Replicate can be a possibility.

———×———

* Hadoop Map Reduce

(steps in slides for implementation)

- Makes large scale parallization very simple.

MapReduce

Map

↳ Works on subset of data & generates intermediate output.

(Input Shard → Outputs (Intermediate))

Once all map tasks are done.

Assumption is map outputs are generally small.

Reduce

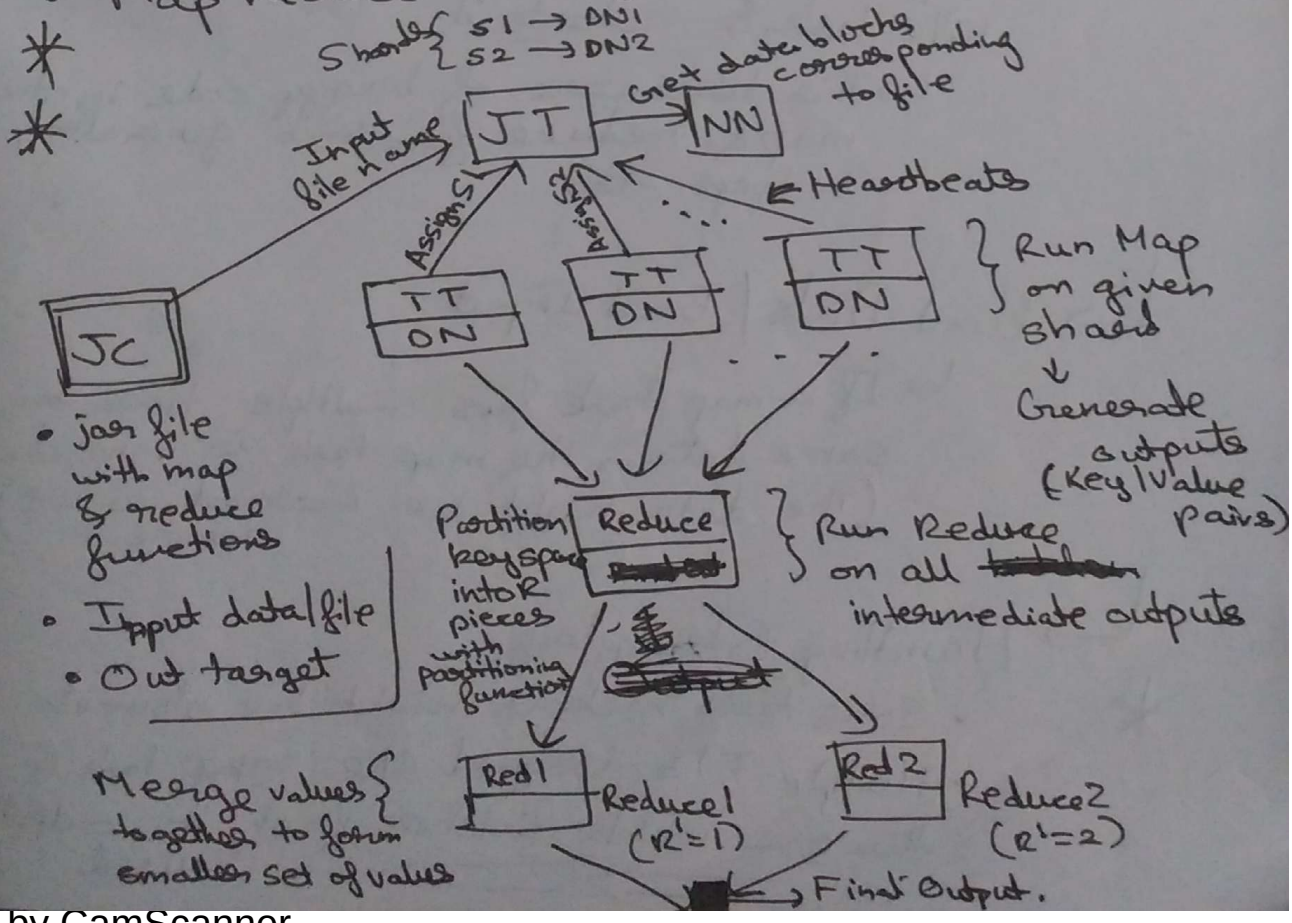
↳ Combines all outputs generated intermediate map/value pairs into result files.

⇒ Launching JVMs takes a lot of time, so we just maintain a pool of JVMs assigning them as needed

⇒ JT = Job Tracker
TT = Task Tracker
JC = Job Client

— x —

- MapReduce is integrated with HDFS



*Handling Failures: (After Failure Detection)

- Note: Attribution
  - ↳ MapReduce is used for this.
- Zoo Keeper → Grid
- Chubby → Google used.

↳ JT dies

* • Single point of failure
  - ↳ On restart we need to know,
    - All job states → map & reduce → started, completed
    - TT's & JC's need to discover and identify where the JT is.
    - Election of job trackers? → Anyone can take up the job.
  - ↳ A solution could be that we store the job statuses on HDFS & the new JT can read these, when TTs find out the new JT & heartbeat their statuses, the JT gets the latest updated state.

↳ (TT dies/fails) / (Task Failures) / Buggy functio

* • JT reschedules whatever this TT was doing to another active T.T.
  - TT launches a new process & execute task, if process fails (no progress in 10 mins), TT kills task & restarts it. ↳ timeout
    - ↳ This takes care of buggy code in the map & reduce functions generating ∞ loops etc.

↳ Bad Disk / Bad Input

- ↳ If a map task fails multiple times on same data, the map task is ignored. (The data might be incorrect, so we ignore it)

↳ Handling Straggles

* • Some task trackers might be slow etc.
  - Multiple TTs assigned the same task & the one which finishe

Note: Zoo Keeper (ZK)

- Highly available shared memory
- Files
  - Permanent
  - Ephemeral
    - If a creator failure occurs, the file disappears.
- Notifications of events

⇒ A simple leader election can be done by the creation of an ephemeral file. Creator who successfully created the file is the leader.

⇒ If a leader dies, all other contesting processes get a notification so they try to become the leader.

Note: Lots of examples of map reduce in slides

- Distributed grep
- Inverted index

Failures

- Having high availability means total failure
* is very low but one of the available systems failing is of a higher probability.

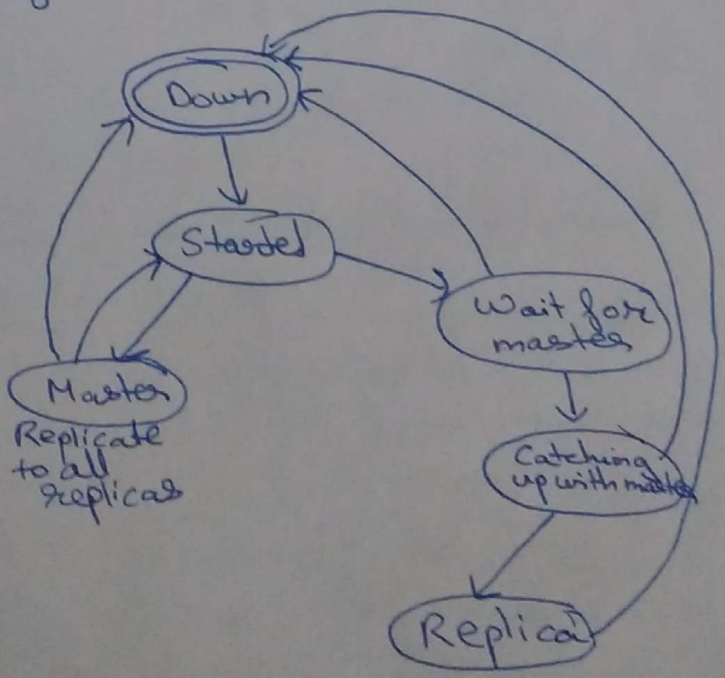
→ Single node → Failure prob = p → No failure = 1−p
→ N nodes → p^n } Very low → $1-p^n$
→ Partial failure = $n \times (1-p)$

---

* MTBF = Mean time before failure
MTTR = Mean time to recovery

⟹ MTTR should be as low as possible

- System is a state machine



- MTTR is time taken to get to Master/Replica from Down state.

⇒ Slaves could be of 2 types (Replicas)

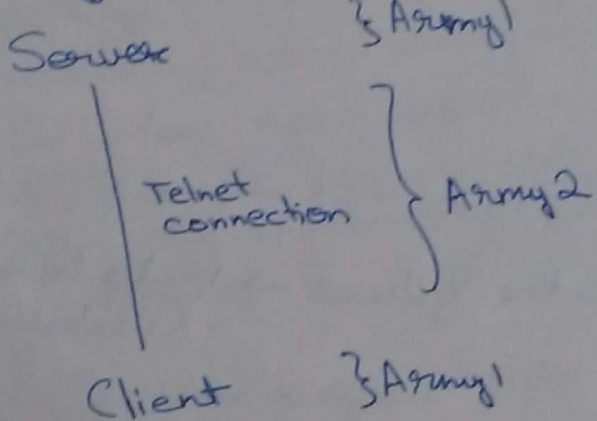
* → Active – passive
  - Slaves just copy writes & are ready to become masters.

→ Active – active
  - Slaves handle queries & reduce load on master.

——— x ———

* Fault Detection:

1). Two army problem:
↓ Translated into networks



⇒ No matter how many acknowledgements are sent between server & client, we cannot determine if they have communicated successfully or not. → Infinite acknowledgement problem

* • In distributed systems it's hard to know if a node is down or it was network issues etc.
  ↳ we just use heuristics like heartbeats, if x heartbeats miss we assume the node failed although it might just be the connection dropping packets.

——— x ———

*Fault Corrections:

1) Heartbeats for detection & data updation

*2) Operation log:

↳ Apply all the updates again in the correct order.

*3) Merkel Trees:

*

Note: Byzantine failures → Nodes lie to you (Hacked nodes etc)

Distributed Transactions

- Atomic, consistent, isolated (serializable) & durable.
  - Atomic → No one sees intermediate results (Single WAL)
  - consistent → Same view for everyone

——x——

⇒ An example of a transaction could be writing to a data node. (In lets say HDFS)

* 1) ↳ One option could be
  - Create duplicate node, write there & then set original pointer to this & garbage collect the original. (This is basically using shadow blocks)
  - So if we dont set the original pointer to this (ie. Rename was not done), than the transaction failed, else it worked.

2) ↳ Write Ahead Log — stores all performed actions.
  - If transaction "END" is present, then its a success, else its a failure.

——x——

⇒ Distributed transactions are where multiple systems take part to complete a transaction.

——x——

Note: * mv /dev/sda/.... /dev/sdb/.... , this cannot be atomic since we generally use a WAL, but since its on different disks, each disk would have a WAL, and this could essentially be called a distributed transaction. (Not atomic)

1) * * Two-Phase Commit Protocol: (Distributed Transactions)

- Reliably agree to commit or abort all sub transactions

⇒ One transaction manager is elected as the coordinator

↳ Asks all the participants if they want to commit or abort. If anyone says abort, the entire transaction is aborted by the coordinator. Else it commits.

- 2PC assumes a failed system will eventually recover.
- Each system stores a WAL.

* (2 Phases, example in slides)

⇒ All participants need to maintain WALs since if the coordinator crashes after participants send commit message, it comes up & asks participant to do the same thing again & they should refer to their log & not perform the action again.

* Failures

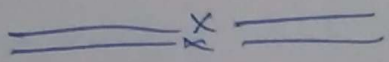
* ↳ Coordinator dies
  - Coordinator checks its log & restarts entire transaction.

↳ Participant dies
  - Comes back, rollback using WAL.
  - Gets request from coordinator again & does its subtransaction.
  - Sends acknowledgement to coordinator

⟹ Failures aren't such a big issue here.

* Blocking Protocol!
  ↳ This is a huge issue
  - If either coordinator crashes or participant crashes, everyone just ends up waiting.
    (A recovery coordinator is used sometimes)

* ⟹ One thing that's necessary is that each transaction ID is always unique (even on restart), so that the coordinator & participant are always talking about the correct transaction (latest).



* 2) Three-Phase Commit (Slides) (Recovery etc in slides)

** a) Voting phase } Similar to 2 phase commit protocol.
   b) Prepare to commit phase
   c) Finalize

- If all participants (in phase 2) send 'OK' messages to commit, then coordinator goes to phase 3, else abort.
- Once in phase 3, participants can talk to each other and the one who goes down & comes back up can just get state from anyone else & then perform commit if required.
  (So even if coordinator goes down its fine)

Issues: Its not completely blocking now, but
* partitioned networks pose issues. (Slides)

Note: MongoDB — NoSQL Database

- Atomic cross row transactions are not possible.
(There are other ways to do this)

—x—

Paxos (Consensus)

- Roles → Proposers (propose a value by contacting all acceptors)
→ Acceptors (accept or reject a proposed value)
→ Learners (wait for a proposal to be accepted)

⇒ Should reach consensus if more than n/2 nodes are up & running.

—x—

* Naive Algorithm:

→ Impose an ordering on proposals
→ Acceptors accepts highest numbered proposal.

- Proposal accepted by majority is a consensus.
→ Issue is, when can we be sure that a consensus has been reached?

—x—

Note: Quorum protocol

** → Obtain response from majority of nodes to make a decision.
→ Bully protocol is not a quorum protocol.
→ Paxos & Raft are quorum protocols.

—x—

⇒ In paxos, we assume acceptors & proposers are different.

- Paxos → Single decree } Only one consensus reached
(Variants like multi paxos etc)

Paxos

Acceptors

State ⇒ $[a, (a', v)]$

Propose: (Request)
- When it gets a propose proposal, (c, v)
- ⇒ If $c \geq a$, send "Yes" and update $a \to c$.
- ⇒ If $c < a$, reject.

(Total = 2p+1 acceptors)

Accept: (Request)
- When it gets a accept proposal, (c, v)
- ⇒ If $c < a$, reject
- ⇒ If $c \geq a$, $a \to c$, $a' \to c$, $v \to v'$

At times acceptors may be down, so they don't receive accept requests & hence their (a', v) may not be equal

$(a', v_{a'})$

⇒ On a propose request if a majority of the acceptors (p+1 or more) set the same 'a' in their state, (majority respond with "Yes") then an accept request is sent to all, with (a, v') where v' would be the value corresponding to the largest 'a' among all the acceptors stored state! (from proposer)

⇒ Suppose acceptors send "Yes" to proposal then they also send their values $v_0, v_1, v_2, \ldots$ (All the acceptors) and then the proposer sends, an accept message $(a, \max(v, v_1, v_2, \ldots))$.

Choose v_x which has max clock.

$v \to$ Clock a's
$v_1 \to$ Clock b_1
:

← Based on stored clock a' of each acceptor

- We do the max operation since (a', v) need not be equal for all acceptors.

Proposers

- Stateless (Just use a clock)

Note: A proposer feels that consensus is reached in the system if more than or equal to N+1 nodes acknowledge the "accept request"

Proof of Safety:

- Claim: If a value v is chosen at proposal n, then

Proof by contradiction:

- A = set of nodes that accepted (n, v)
  B = set of nodes that agreed propose (N)

→ Since $|A| \geq$ T+1, $|B| \geq$ T+1 (T = Total acceptors), there will be atleast 1 element intersection.

- $p \in A \cap B \rightarrow$ p sent back (K, v), $K \geq n$ with p's state = $[K, (K,v)]$ "Accept"
- Consider some $q \in B$, it sends $[m, v]$, but since $m > K \geq n \Rightarrow m > n$
  → But this is a contradiction since p still has (K, v) where $K < m$.

- Conflicting proposals, accept requests, proposers proposing new values after agreement is reached
- Acceptors going down after consensus is reached

* are an issue.

→ A_1, A_2, A_3, A_4, A_5

Accept and reach a consensus

→ A_1, A_2 die, so A_3, A_4, A_5

A_3 Reached consensus

→ Now if A_4, A_5 accept another proposed request since they have a majority now, they would be having another consensus.

⟹ $\therefore A_1, A_2, A_3 \longleftrightarrow A_4, A_5$ } Both proposers feel they have consensus leading to issue

---

* Hierarchical Systems:

- Z_1 Z_2 Z_3 } Paxos used to elect first leader at this level

K_1 K_2 K_3 } Prev leader used to select a leader at this 2nd level

- These could be used to automatically get a hierarchy of leaders who can be used for various tasks.

- On response
  - Success → Become Leader
  - Timeout (N+1 not received) → Increase term number (Retry after timeout)
  - Leader responds (Saying that they are already the leader so stop) → Become Follower

If not randomized everyone would vote for themselves in each term & no-op.

This would be randomized so issues won't arise.

⇒ This would ensure that for a single term there would be only one associated leader node.
(An issue might occur that 2 nodes are leaders at the same instant with 2 term numbers)

* ↳ This is fixed with heartbeats
  - Every leader heartbeats others & if someone responds saying they are the leader at a higher term. The current leader drops out.

(Note: There can be terms with no leader elected successfully)
(These timeouts are also used to detect failures)

*2) Log Replication: (Done using heartbeats → RPC Append Entry (Term, Log #, entry))

- Leader gets an update

⇒ Forwards update to all followers, if majority of them ack then send ack to client who sent [redacted] the update.

- ↳ Ack'd updates are written to the log.
- If leader doesn't get ack from majority, keep retrying..

Note: Followers logs can be different (but they will be a prefix of leaders log) → This is not true, we

- Followers are sent data along with log index & term number. The follower updates their log at the log index.

(Log entries – a variety of states for leaders & followers could be possible (example in paper)

* ⇒ Along with the data, log index & term number leader also sends the previous entry data, log index & term number. So now if replica's prev entry matches it updates the new entry, if it doesn't match ask the leader again but this time for the previous entry. (So leader sends previous-previous entry & the previous entry)

- In this manner we can go back to the point where the replica & leader branched off & then replica copies the correct state from the leader.

*3) Fixed Leader Election:

- When you get an election request, the requestforvote will also contain the latest term, log index of the requester.

⇒ Now if your term < term$_{req}$ or term = term$_{req}$ & log index < log index$_{req}$ then send "Yes" else reject the request.

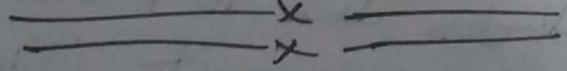
- In this manner the elected leader would be one who has all the latest writes (committed)

⟹ Proof for this is via the pigeon hole principle.

↳ A node commits (current leader) then atleast n+1 nodes have the latest update.

↳ If an older node tries to become the leader then it can get at max only 'n' accepts so it can't become a leader.

↳ Now if the leader makes another commit, then atleast n+1 nodes get updated. So nodes having all latest updates = Intersection ≥ 1. So atleast one node is capable of becoming the leader.



Deadlocks

ex: Transaction from A→B & B→A simultaneously

```
start()                          start()
lock(A)                          lock(B)
lock(B)                          lock(A)
A += val                         B += val
B -= val                         A -= val
unlock(A)                        unlock(B)
unlock(B)                        unlock(A)
end()
```

If we perform till here in both then its a deadlock.

- The solution for this is to ensure that the locking order is always the same. (Banker's Algorithm)

———x———

⇒ Deadlocks due to limits on network resource usage etc. (Similar to the above issue)

$P_1: A \to B \to C$, $P_2: B \to C \to A$, $P_3: C \to A \to B$ (we can lock one link at a time)

———x———

Wait-For Graphs:

- Resources (R)
- Processes (P)

$R_1 \to P_1$ } P_1 holds R_1

$P_1 \to R_1$ } P_1 waits for R_1

⇒ Any cycle in the directed graph indicates that there can be a deadlock.

———x———

*Deadlock Detection:

1) Centralized Deadlock Detection:

- Each node maintains its own wait for graph.
- A centralized coordinator would get all these graphs and construct the global wait-For graph.

* ↳ An issue that occurs is if the messages arrive out of order.
  - The solution to this could be to have a global ordering

ex: $R_1 \rightarrow A$ $R_2 \rightarrow B \rightarrow R_1$

*
- If we get A wait R_2, B release R_2 although original order is the reverse. then, (Failure case)

$R_1 \rightarrow A \rightarrow R_2 \rightarrow B$ } Deadlock.

- Otherwise

$R_1 \rightarrow A$ $B \rightarrow R_1$

R_2

$B \rightarrow R_1 \rightarrow A \rightarrow R_2$ } No Deadlock

⇒ Or a simpler solution is that the coordinator asks all the nodes if they have any pending release messages & apply them first.

—— x ——

*2) Chandy-Misra-Haas:

* • When a process tries to acquire a resource but it cannot, it sends a message to everyone saying {blockedID, myID, receiverID}

⇒ So if P_3 is waiting for R_1 where P_1 has it already held, it sends the probe message keeping blockedID = P_3, myID = P_3 & receiverID = P_1.

- Now if a node receives a message, it forwards to all processes that have held a resource it is waiting for, keeping blockedID the same and changing myID & receiverID as needed.

⇒ If a node gets a message where the blockedID is equal to the nodeID then there is a deadlock. (Basically cycles get detected)

---

*Deadlock Prevention:

- The only thing we can try to avoid is the circular wait, between processes.

* ⇒ Assign timestamp to each transaction
  - Newer transactions have higher numbers.

⇒ Transactions with smaller numbers are given the lock first. (Or vice versa).

- Kill the transaction & restart if we feel a deadlock is being detected.

1) Wound-Wait:

2)

———— x ————

Note: In practice simple heuristics to detect deadlocks works really well. ex: If transaction does not complete successfully in x minutes, assume its a deadlock.

———— x ————

*Scaling Transactions:

*

- CAP theorem → Consistency
  → Availability → Accessibility
  → Latency
  → Partition tolerance (failure handling)

⇒ Use replication to scale.

→ Replicas cache data & handles reads
→ Keeping replicas consistent → Distributed transactions
Becomes hard.

———x———

Brewer's CAP Theorem → You can only have almost 2 out of CAP.

———x———

ex: RAFT mainly had C & P. Low availability since all writes → N+1 nodes etc.

P → <N+1 nodes fail, both C&A.
→ >N+1 nodes fail, lose A.

- Reads handled by master → So availability is low.

———x———

Note: We move on to eventual consistency since complete consistency is quite hard.

→ BASE instead of ACID.

———x———

⇒ In general nowadays everyone uses eventual consistency → Facebook etc. which is why at times the data is stale.

*Consistency:

*1) Client Centric Consistency:

*. • Monotonic reads: All reads should be giving more recent updates as we move forward in time.
(example of inconsistent behaviour in stale)

• Monotonic writes: Writes to a certain variable from a same process go to all replicas in the same order.

• Read your writes: Effect of a write by a process will be visible to all further reads by same process.
(other processes might not but that is fine)

⇒ Generally replication of data b/w servers is done using asynchronous replications.

— x —

ex: Raft, we are always sure that 'Read your writes' consistency is satisfied since an elected leader always serves reads and elected leaders have all updates.

— x —

⇒ Read Your Writes: (Consistency)

*. • To do this when client contacts a server & does a write, it gets returned a version number (clock). Now on reads keep pinging the servers until the version numbers ≥ stored version number, to get consistent data.

⇒ This would require the client to be stateful. The version numbers would be need to stored in memory.

— x —

Big Table
(Google)

- Bigtable is a sparse, distributed, persistent multidimensional sorted map.
- Has rows & columns → columns are dynamic not all initially populated.

* ⇒ Sorted rowwise by row key.

---

ex: rowkey = URL, columns consist of images on page, text size on page .... etc.

- Each cell identified by row, column but the cell also stores previous entries. ∴ (row, column, ~~content~~ timestamp) → value/cont

* Tablets:

- Row operations are atomic

⇒ Tablet = Range of contiguous rows

! • Selecting good row keys is important

* ex: Reverse domain names since we can
* move among all 'com' websites since they would be grouped near each other. (To get temporal locality (nearby rows))

⇒ Since nearby rows are served by the
* same tablet server, its fast. (Prefetch optimizations etc)

Note: NoSQL databases are famous since schema
* changes, semistructured data, key→value stores and scalability is not accounted for in relational databases. (RDMS → ACID, No-SQL → BASE)

⇒ Initially entire table is just a single tablet.

* Tablet Splitting:

- As more rows are inserted, table is split into tablets, each tablet has start & end row keys. (Around 100-200 MB each tablet)

⇒ Column Families: Groups of columns in sparsely populated data.

a) An example is Google Analytics

RawClickTable → (With MapReduce tasks. (Periodically)) → SummaryTable

RawClickTable
{website name & time of session}

b) ⇒ Another example is Personalized Search (Google)

* Implementation:

- BigTable was built upon GFS (Google File System)
  - ⇒ GET, PUT requests on DB. (Similar to HDFS)
  - ⇒ Need to maintain a WAL.
- WAL needs 'append' on HDFS, so this was added to HDFS. (Only append not complete replace)

Note: We use sorted key ordering instead of maintaining hashes etc, since addition of new node requires rehashing all existing

ex: HTML documents etc don't have a fixed structure, so for these kinds of data, No-SQL is useful.

- SSTables were implemented which stored results till a certain point in the WAL.
- Initially requests are stored on memory and after a while its written to SSTable (Sorted String Table) and now this is persisted on system reboot and we don't need to apply the WAL all the way from the beginning.

⇒ All SSTables are stored in HDFS.

* Now on a GET request we search the memory first and if its not there we start and go through the most recent SSTable and then keep going backwards until we find the key. (Since SSTables are sorted, we can do things like binary search etc)

* • BigTable → SSTables (+Index Files) + WAL + Memory (MemTable)
* * (Breakdown of the components) — To speed up reads — To speed up writes

Note: Since we go backwards on SSTables, we would get the latest updates (PUTs) first and we return the latest value of a given key. (The old garbage data would still be unnecessarily present)

* (But this is also required since we query with timestamp (row column))

(One index file per SSTable)

Note: Index files are small & stored in memory. It has key, offset in SSTable pairs. So if we find our key, we directly read from that point in the SSTable (corresponding)

* Bloom filter: (Instead of Index files)

- When we ask if a key is there
  - Yes → May be present. (False positives could be there)
  - No → Definitely not there

⇒ Some implementations use these bloom filters. whereas others use index files.

———— x ————

Note: Major compaction is done every now and then

* • Create a new SSTable + Index file from older SSTables. (This happens in the background)

———— x ————

• Deletion of keys is quite simple, we just add
* a column saying 'Deleted' and mark it as true. (So deletion is as fast as a PUT request, GET requests would have to handle this column in a special manner).

⇒ Major compaction takes care of freeing up space by removing very old SSTables etc.

* * Failure Handling:

- Master server and multiple tablet servers. Each tablet server has all the Index Files, WAL & its MemTables.

⇒ If a tablet server dies, master allocates a new server for this tablet and it updates its data. Similarly if tablet server becomes too large, the master splits it into two servers. (i.e. if many SSTables in HDFS are taken care of by a single tablet server, then GETs to that server would be delayed, so they are split. The actual files however don't need to be changed since every node uses same HDFS)

———— x ————

* BigTable offers consistency (not eventual), so reads should go to same tablet servers after writes since the memtables are unique for each tablet server.

⇒ To achieve this we use a Meta Table;

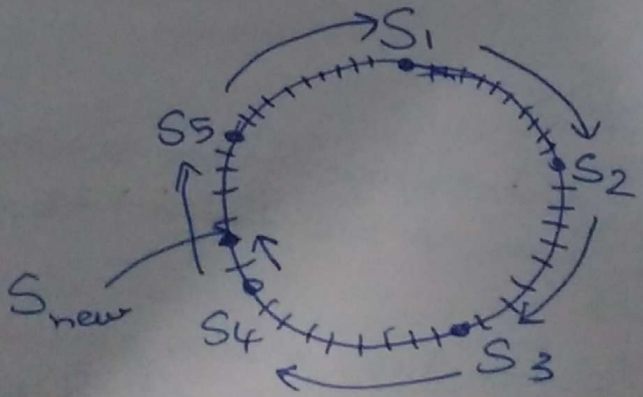
↳ Key ⇒ (Tablename:Key), Value ⇒ Tablet Server

- So when we get a 'GET' request, we check this meta table and send the request to that Tablet Server.

---

* Consistent Hashing: (Chord)
*
(Used in Dynamo)

- When servers are added, instead of having to rehash all keys, we try to reduce this operation.



- For a given key, it is served by the nearest server in the clockwise direction

⇒ On adding new server Snew, only the few keys between Snew & the server before it need to be updated.

Dynamo

(Section 4 of paper) (Amazon - Key Value store)

- Goals: (Eventual consistency)
  - * → Low latency → Configure Availability.
  - * → Always writable (even if a single node is up)
  - → Geographically spread nodes
  - → No centralized server.
  - → (Fully decentralized)

- Accessed by primary key → Key-Value store.
- N → No. of total nodes (2N+1)
- W → No. of nodes that write. (N+1)
  R → No of nodes to read from (1)

⇒ On a write request, keep performing * writes as we move clockwise along the ring till we get W acks. Similarly for reads.

- Consistent Hashing ensures we get the same node if the total number of nodes in the system is the same, else it would give a different server hash.

i) * Addition of nodes: (servers)

- A new node gets added and using gossip protocol, lets all the other nodes know eventually as each node tells some other nodes picked at random about this new node and the process continues.

ii) ⇒ Some issues (to take care of)

1) Load Balancing

2) Heterogeneous nodes. (More load given to powerful machines)

* These were taken care of by mapping a single node (machine) to multiple spots on the ring. (So it gets a larger range on the ring)

iii) ⇒ Get & Put requests:

* Put replicates to neighbours in the ring. Given a put(k), k gets hashed to some place on ring and then moving clockwise first node does the put along with w replicas in that direction. (If nodes are down then we skip over)

⇒ For every key k we have a pref list, this says which nodes should be read for that key k. (Each node has its own pref list for the same k)

* We also use a vector clock for reads.

↳ If a node gets an update then it increases its portion of clock and sends this to all the nodes where it replicates. These nodes update their clocks based on the senders clock.

* Now on Get requests for all those R nodes, we take all their clocks and perform some comparisions.
(Before a put as well, we do a get and then update the content based on the response of get) (These get requests generally give all same values but at times they differ)

eoc: For a shopping cart application suppose 2 read responses from different nodes have different values, we just consider union of those since showing additional things in shopping cart once in a while is fine.

———x———

iv) Hinted Handoff:

* If a node is down, then the next node stores the requests that the node was supposed to handle.
* When the node is back up and this node gets to know during gossip protocol, it hands over the requests to it.

v) Divergence: (Not that common)

* If the same key is requested (put) by different nodes then its called divergence.
* Divergence occurs if initial serving node of key is down or the new client putting put(key) does not know of initial serving node.

———x———

Note: No matter what R & W we pick, in the worst case we would not get perfect consistency. (Unlike in raft where if R+W>N then its consistent).

(eoc: 2 Phase commit, W=N & R=1)

(Dynamo allows us to tweak W & R which is great)

———x———

———x———

Note: Cross row transactions in MongoDB, BigTable

* i) Append in HDFS.

——— * ———