Jin Wei CS4224 Distributed Databases

Data Partitioning

Desirable properties of fragmentation

- completeness: each item in R can be found in ≥ 1 fragment (nothing is lost)
- reconstruction: R can be reconstructed from fragments (must be lossless join)
- disjointness: data items are not replicated

Fragmentation Techniques

- Range Partitioning
  - Use predicates on ≥ 1 attribute (e.g. < 100, $[100, 500)$, ≥ 500)
  - Use catch-all predicate to guarantee correctness OR use a binary tree (p_1 vs $\neg p_1$)
- Hash Partitioning
  - Good hash function + not-skewed data => data will be evenly distributed
  - Method 1: modulo method - when adding / removing nodes: need to rehash everything (less elastic)
    - In R_i if $h(...) \mod i$ is i
  - Method 2: consistent hashing - easier to add / remove nodes (more elastic)
    - Partition $h(...)$ using n values into $v_1 < ... < v_n$ (replicated to all nodes OR just master)
      - 1st: $\leq v_1$ **and** $> v_n$, 2nd: $(v_1, v_2]$
    - In R_i if $h(...)$ is in R_i's range, R_1 is catch-all node
    - Non-uniform data & load distribution (can be managed with virtual nodes -> let variable amount map to same physical node)

Derived Horizontal Fragmentation

- Partition a R based on S using semi-join $R_i = R \ltimes_A S_i$
- For completeness, $R.A \subseteq S.A$
- For disjointness, $S.A$ must be key
- So for both, R.A must be FK of S **and** NOT NULL

Vertical Fragmentation

- key(R) must be in all partitions (for disjointness: only keys are duplicated)
- heuristic Attribute Affinity Measure: if commonly referenced together, should be in same partition

Complete Partitioning wrt Query

A partitioning F is **complete** wrt to Q if for all fragments $R_i \in F$: either return WHOLE partition *OR* nothing.

Minterm Predicate Partitioning

Minterm predicate m = a combination (positive / negative) $\wedge$ for a set of predicates ($|m| = 2^n$). Use boolean algebra to simplify

- $Q = \{Q_1, ..., Q_k\}, Q_i = \sigma_{p_i}(R)$
- $P = \{p_1, ..., p_k\}$
- $F = \{R_1, ..., R_m\}$ where $R_i = \sigma_{m_i}(R)$

Minterm predicate partitioning F (of Q) is always a complete partitioning wrt to the Q. Watch out for $\vee$.

Query Processing

Make query plan that minimises total cost (CPU, I/O, comm) OR response time. Try to parallelise queries.

1. Normalisation (rewrite query into normal form)
   - *more common CNF $(p \vee p) \wedge (p \vee p)$
   - DNF $(p \wedge p) \vee (p \wedge p)$
   - p is simple predicate: single attribute A_i op v
2. Semantic Analysis (check against schema + type check)
3. Simplification & Restructuring

Localisation Program

- Rewrite distributed query into fragment query
- $\cup$ for horizontal partition; $\bowtie$ for vertical partition

Reduction Techniques

Identify & remove queries that do not contribute to result

- Reduction with Selection: $\sigma_p(R_i) = \emptyset$ if $R_i = \sigma_{F_i}(R)$ and $F_i \wedge p$ = false
- Reduction with Join: $R_i \bowtie_a S_j = \emptyset$ if there's no "intersection" of predicates on *join* attributes (a)
- Reduction with Derived Frag.: $S_j \bowtie_a R_i = \emptyset$ if S_i is derived from R and $i \neq j$
- Reduction with Vertical Frag.: if missing required attribute, drop the fragment

Distributed Join Strategies $R \bowtie_A S$

1. both R and S are partitioned on join key
2. only R (not S) is partitioned on join key
3. neither are partitioned on join key

Communication Cost

- Collocated: 0
  - all servers perform local join -> send results to server
- Directed: $size(R)$ if R is repartitioned (R is NOT previously partitioned)
  - repartition -> if in wrong server, send to correct one
- Repartitioned: $size(R) + size(S)$
- Broadcast: $(n-1) * size(R)$ where R is smaller one
  - broadcast smaller table to ALL servers

Storage

LSM (Log-Structured Merge) Storage

Writing to B+ is random I/O (+ splitting & propagate); use LSM instead (append-only updates)

- Memory Table (in memory hash table with in-place updates)
- After threshold: sorted + flush to disk (sequential I/O)
- SSTables (Sorted String Table): immutable; records are sorted by K; each SSTable associated with *range of key values + timestamp*
- Commit Log Files used for durability

Compaction of SSTables

- Why?
  - improves read performance by defragmenting table records
  - improves space utilisation by removing tombstones (must ensure all other versions are gone) & stale values
    - can remove at bottom-most level (because guaranteed not in any other SSTable at this level, and no higher levels)
- Size-tiered Compaction Strategy (STCS)
  - Each tier has approx. same size
  - compaction triggered at tier L if number of SSTables == threshold
    - All SSTables at L are merged into *one* SSTable at $L+1$
    - Tier L becomes empty
  - Each object has ≤ 1 version in **every** SSTable
- Levelled Compaction Strategy (LCS)
  - SSTables at level 0 can have overlapping key ranges
  - For level ≥ 1
    - each SSTable is same size
    - key ranges do not overlap within the tier
    - SSTable at L overlaps with at most F SSTables at $L+1$
  - Lower level (across tiers) + larger index (within tier) is more recent
  - Compaction: new tables stored at $L+1$, old tables removed
    - for $L \geq 1$: choose a SSTable (round-robin style with wrap-around) -> merge with all overlapping SSTables at $L+1$
    - for $L == 0$: merge all SSTables at level 0 with all overlapping SSTables at level 1
    - if inserting into SSTable violates its F condition, make new table
    - triggered when number of level 0 reaches threshold; for $L \geq 1$: $size(L) > F^L$ MB
  - Each level stores F times as much data as previous
    - For n records of m MB each, in worst case: last level stores a version of each of the n records.
    - Therefore, $F^{L-1} < mn \leq F^L \implies L = \lceil log_F(mn) \rceil$
  - Each object has ≤ 1 version in **every** SSTable (in level 0), has ≤ 1 version in **every other level**.

Searching LSM

- Start at MemTable; go to next level, start at right-most table
- Check key range first: if within, use binary search
  - At each level ≥ 1: either search 0 or one tables

Optimising SSTable Search

SSTables are stored in blocks.

1. Sparse index
   - $(k_1, k_2, ..., k_N)$ if N blocks
   - k_i is *first key value* in block i
   - Binary search the sparse index in-memory
2. Bloom filter
   - if match *ALL* hash functions, *might be* in block (false positive)
   - else, definitely not

Indexing

- Local Indexing
  - Each node stores index for its data
  - Easy to update; still need to scatter-gather
  - Have to check **all** nodes if not partition key
- Global Indexing
  - Index the entire DB, partition index with hashing into all the nodes
  - Hard to update (need another server); good for searching
  - Have to check **single** node even if not partition key

DynamoDB

- item in table has PK, otherwise it's schemaless
  - Single PK = (partition key)
  - Composite PK = (partition key, sort key)
- Tables are partitioned by *partition key*
- Items in same partition are sorted by *sort key* (if given)
- Global Index: index key is simple or composite; partition key can be different from table PK
- Local Index: index key is composite only (sort key is indexed key); partition key must be same as table PK

Distributed Commit Protocols

- ACID
  - Atomicity: all or nothing
  - Consistency: if each xact is consistent and DB starts consistent, it ends consistent
  - Isolation: Executions of xacts are isolated from each other
  - Durability: if a xact commits, its effects persist

Centralised DBMS Recovery Manager

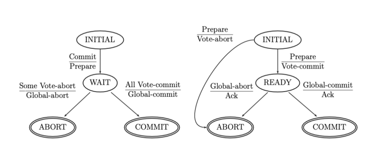
Recovery Manager ensures atomicity and durability.

- Implementing Abort
  - Write-Ahead Logging Protocol: flush uncommitted update *after* before-image is flushed
  - Undoes all updates by xact by restoring before-image
- Implementing Commit
  - Force-At-Commit Protocol: commit a xact *after* after-images of all updated pages are flushed (write commit log, immediately flush)
- Implementing Restart
  - Redo phase: redo all updates
  - Undo phase: abort all active xacts

Failures in DDBMS

- Site failures
  - fail-stop model: a site is either operational OR failed
  - Partial site failure: some sites are operational, some have failed
  - Total site failure: all sites have failed
- Communication failures (all sites operational)
  - lost messages, network partitioning (split-brain problem)

Two Phase Commit (2PC): voting + decision

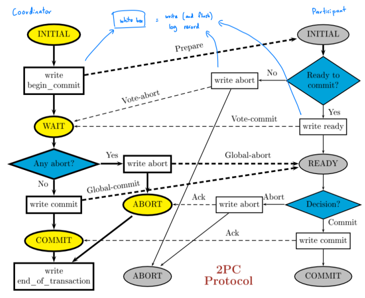


(one state transition apart)

- ALL reach same global decision
- once voted, cannot change vote
- one abort, global = abort
- anyone can abort without voting
- commit => all voted commit
- no failure + all vote commit => commit

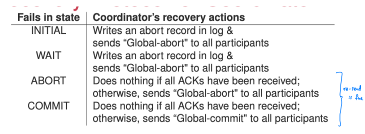
Log records are flushed / forced / synchronous writes

- Coordinator: Force write commit log record, don't force write abort log record
  - (both) Recovers in WAIT: after timeout, will abort
- Participant must force write *ready log record*
  - Recovers in INITIAL: will abort (but global might be commit)
- Participant: Force write *commit / abort log record* (if voted commit), don't need to force write (if voted abort)
  - (vote commit): recovers in READY, will revote commit, coordinator might not be able to inform global decision (because too long later)
  - (vote abort): recovers in INITIAL, will abort -> OK!



Site Failures

- detected by timeouts
- Recovery Protocol (by server that failed)
  - **independent** if can terminate without outside info
- Termination Protocol (by TC)
  - **non-blocking** if can terminate without waiting for recovery



Fails in state	Coordinator's recovery actions
INITIAL	Writes an abort record in log & sends "Global-abort" to all participants
WAIT	Writes an abort record in log & sends "Global-abort" to all participants
ABORT	Does nothing if all ACKs have been received; otherwise, sends "Global-abort" to all participants
COMMIT	Does nothing if all ACKs have been received; otherwise, sends "Global-commit" to all participants

Fails in state	Participant's recovery actions
INITIAL	Aborts transaction unilaterally
READY	Sends "Vote-commit" to coordinator
ABORT	Does nothing
COMMIT	Does nothing

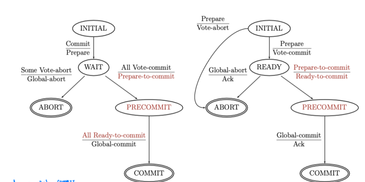
Timeout in state	Coordinator's termination actions
WAIT	Writes an abort record in log & sends "Global-abort" to all participants
ABORT	Sends "Global-abort" to participants who have not responded
COMMIT	Sends "Global-commit" to participants who have not responded

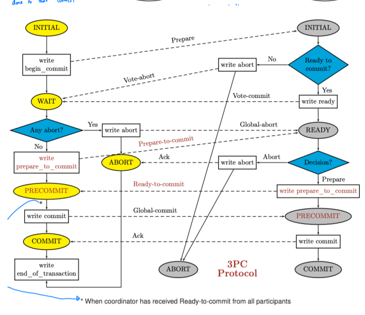
Timeout in state	Participant's termination actions
INITIAL	Aborts transaction unilaterally
READY	Participant is blocked!

Cooperative Termination Protocol

- Let participants communicate with each other
- If P timeouts in READY: P asks for decision
  - if *any* node is COMMIT / ABORT: P does that & tells anyone who is READY
  - if a node is INITIAL: it aborts, and replies ABORT
  - if all are READY: blocking!

Three Phase Commit (3PC)





- Termination Protocol Changes
  - Coordinator's pre-commit: write commit log record + send global-commit to operational participants
  - Participants' ready + pre-commit: execute Termination Protocol X
- Recovery Protocol is same as 2CP

Termination Protocol 1

- Protocol
  - Run leader-election to elect C
  - C requests state from participants
    - if *any* in COMMIT: Global-Commit to all
    - if *none* in PRECOMMIT: Global-Abort to all
    - else: Send Prepare-to-Commit to READY, receive Ready-to-commit from these, then Global-Commit to all
      - 2 steps are needed: otherwise if it crashes again and none in PRECOMMIT, second `if` is triggered
  - if any participant timeout (coordinator failed), elect new coordinator; any participant that fails is ignored; anyone that fails and recovers **CANNOT** participate
- Total Site Failure:
  - Recovering TMs blocked until a TM P recovers:
    - P can recover independently (state = INITIAL, ABORT, COMMIT)
      - P notifies recovered TMs
    - P was last TM to fail
      - P executes termination protocol
- Without total site failure + comm. failure: non-blocking
- With total site failure: blocking
- With comm. failure: might be split-brain

Recover from earlier failure cannot rejoin: single node in PRECOMMIT, Coordinator & that node fail -> global will be Abort (but if he rejoins, it should be commit)

Termination Protocol 2 (handles comm. failure)

- Protocol
  - Run leader-election to elect C
  - C requests state from participants
    - if *any* in COMMIT: Global-Commit to all
    - if *any* in ABORT: Global-Abort to all
    - if *any* in PRECOMMIT + *no* COMMIT / ABORT + majority in READY / PRECOMMIT: Prepare-to-Commit to ~PRECOMMIT; receive Ready-to-commit
      - if number of PRECOMMIT + Ready-to-commit is majority: Global-Commit
      - else: blocked
    - if *no* COMMIT / ABORT + majority in INITIAL / READY / PREABORT: Prepare-to-Abort to ~PREABORT; receive Ready-to-abort
      - if number of PREABORT + Ready-to-abort is majority: Global-Abort
      - else: blocked
    - else: blocked
- Blocked TMs periodically re-attempt protocol; when failed TM recovers: executes protocol
- Non-blocking as long as majority are operational

Concurrency Control

Concurrency Control Manager ensures isolation.

- VE if same read-froms & same final-writes
- VSS if VE to some serial schedule
  - VSG - (T_j, T_i) if T_i reads-from T_j, or both write to same variable and T_i does final-write
  - VSG cyclic => *not* VSS
  - VSG acyclic & (serial schedule from topo-sort is VE to S) => VSS
- conflicting actions if
  - at least one of them is **write action**
  - and actions are from different transactions
- CE: every pair of conflicting actions are ordered in the same way
- CSS: CE to some serial schedule (CSS => VSS)
  - CSS $\iff$ CSG is acyclic
- blind write: Xact no read before it writes
  - VSS & no blind writes => CSS
- Recoverable Schedule (essential): for every Xact T that commits in S, T must commit after T' if T reads from T

Lock-Based CC

Requested	Held: -	Held: I	Held: S	Held: X
I	Y	Y	N	N
S	Y	N	Y	N
X	Y	N	N	N

- if lock request not granted, Xact is blocked, Xact is added to O's *request queue*
- 2PL => CSS: once release a lock, no more request
- Strict 2PL => strict & CSS: Xact must hold onto lock until commit / abort
- Wait-For-Graph: $T_i \rightarrow T_j$ if T_i waiting for T_j (must remove edge)
- Timeout mechanism: when Xact start, start timer, if timeout, assume deadlock
- Deadlock Prevention - older Xact has higher priority (not restarted on kill to avoid starvation)
  - suppose T_i requests a lock held by T_j (Higher; Lower)
  - wait-die (non-preemptive): T_i wait for T_j; T_i suicide => may starve
  - wound-wait (preemptive): kill T_j; T_i wait for T_j
  - if T_j dies, T_i still waits

MVCC (multiple ver.)

- read-only are never blocked / aborted
- update xacts not blocked by read-only xacts
- MVE if same read-from
- MVSS if MVE to some serial monoversion schedule
  - monoversion: each read action returns the most recently created object version
  - VSS $\subseteq$ MVSS (not other way round)

Snapshot Isolation (MVCC protocol)

- SI: Xact T takes snapshot of *committed* state of DB at start of T
  - can't read from concurrent Xacts
  - Concurrent if overlap start & commits
  - O_i is more recent than O_j if T_i commit after T_j
  - Concurrent Update Property: if multiple concurrency Xact update *same object*, only one can commit (if not, may not be serialisable)
- First Committer Win (FCW): check at point of commit
- First Updater Win (FUW) - locks only used for checking (**NOT** lock-based)
  - to update O: request X-lock on O; when commit / abort, release locks
  - if not held by anyone:
    - if O has been updated by concurrent Xact: abort
    - else: grant lock
  - else: wait for T' to abort / commit
    - if T' commit: abort
    - else: use (if not held by anyone) case
- Write Skew Anomaly (not MVSS)
  - Both Xact read from initial value
- Read-Only Xact Anomaly (not MVSS)
  - A Read-Only Xact reads values that shouldn't be possible
- SSI (Serialisable SI): produced by SI and is MVSS
- Garbage Collection: delete version O_i if exists a newer version O_j st for every active Xact T_k that started after commit of T_i, T_j commits before T_k starts (aka all active Xact can refer to O_j)

Distributed CC

- Global schedule S for T and $\{S_1, ..., S_m\}$ is VSS / CSS if
  - Each local S_i is VSS / CSS
  - and local serialisation orders are compatible
- Centralised 2PL (C2PL)
  - all locks are managed by central TM's lock manager
- Distributed 2PL (D2PL)
  - each site manages locks for their own stuff

Distributed Deadlock Detection

- Centralised approach
  - Each site maintains local Wait-For-Graph

- One site is Deadlock Detector: others periodically transmit local WFG to it

Centralised SI

- one site is Centralised Coordinator (responsible for assigning timestamps)
- assume FUW & write locks are distributed
- Performing Transaction
  - TC requests CC for `start` and `lastCommit`
  - for read X, TC requests TM_A to send most recent X wrt `lastCommit`
  - for write X, TC requests TM_A; might be blocked
- For committing xact T:
  - TC requests CC for `commit`
  - TC executes modified 2PC
    - in voting phase: TC includes `start` & `commit` in PREPARE messages
    - when participant receives PREPARE, it checks for WW-conflicts between T and committed concurrent xacts (votes abort if any)

Producible?

Need to check S2PL & SI in local schedules, and then consider global schedule across sites.

- S2PL: not global CSS => not S2PL; check for S2PL locally
- SI: if each object updated by at most one xact (xacts have disjoint write-sets) => can be SI
  - else: need to consider global schedule across sites to check for concurrency

Data Replication

Improve system availability + performance + scalability

- one-copy serialisable (1SR) if same effect as one-copy DB
  - equivalence: same read from, weak final write (only one replica needs to be the same)
  - checking 1SR: similar to globally serialisable,
    - but need to check read from initial
    - and check *"in-between"* writes (e.g. W_1, R_3, W_2)
- strong mutual consistency if all replicas have identical values

Updating Replicas

- Statement-based replication: forward all SQL statements
- Write-ahead Log (WAL) shipping - file-based OR record-based (streaming)
  - Physical replication: specify page & byte offset to change
  - Logical replication: one logical log record for each affected tuple
- Application-level replication
  - Use triggers & stored procedures; more flexible but higher overhead

Replication Protocols

Assume S2PL + statement-based replication

- Eager (sync) update - **always 1SR**: updates all replicas within xact
- Lazy (async) update - **may not be 1SR**: updates only one replica now; rest async via Refresh Xacts (need to be in-order)
  - probably not 1SR because xacts won't read from one another (will read from initial state)
- Centralised techniques: applied to *master copy* first before propagating to other *slave copies*
- Distributed techniques: update is applied to *any copy* first

Note: reading local is best; when writing to all nodes, need to write to self too

Quick identification:

- Refresh xacts => lazy
- For *some* object, if read & write in **different** servers => centralised (always write in master site)

Eager Primary Copy (Centralised)

- For each *object*, one copy is **primary** (different objects can be in different sites)
- Each master site runs *lock manager* for logical object (for all replicas)
- For read: req lock from master site, then read from *any* node
- For write: req lock from master site, then write to all nodes

Eager Distributed

- Each site runs *lock manager* on physical object (for local replica)
  - Need to request for S-lock and X-lock (blocks if not available)
- Deadlocks are more common
- For read: req lock + read from *any* node
- For write: req lock + write to all nodes

Lazy Single-Master (Centralised)

- *Single master site* for **all** objects
- For read: req lock from master, then read from *any* node
- For write: req lock from master + write to master **now**
- For commit: informs master, master releases locks, gets X-locks for refresh xacts & propagates async
  - Must be in **same order** at all sites, ordered by commit timestamp
  - However, doesn't guarantee 1SR, even with single object + single xact (because don't read your own writes)

Lazy Distributed

- Each site runs *lock manager* on physical object (for local replica)
- For read: req lock + read from *any* node
- For write: req lock + write to *any* node
- For commit: release locks, propagates async (other sites grant X-lock for themselves)
  - Inconsistent updates: requires reconciliation
    - Last-Writer-Wins heuristic: accept the last refresh xact (**but** only correct for blind writes)

Handling Failures

- detect failures using use timeout mechanism
- Slave Failure
  - Lazy Single-Master: synchronise unavailable replicas later
  - Eager Single-Master: change ROWA to ROWAA (Read-One / Write-All Available)
    - Synchronise unavailable replicas later
- Master Failure
  1. Wait for master site recovery (bad for availability)
  2. Elect new master site (must have majority)

CAP Theorem

- Data Consistency
- System Availability
- Tolerance to Network Partitions

When there's partitioned network:

- forfeit consistency: resume execution on selected partition (might be inconsistent if partition does not have old master as it lacks latest update)
- forfeit availability: wait for network to recovery

Quorum Consensus

Handles fault tolerance without replicated state machine.

- Each copy has non-negative weight $Wt(O_i)$
  - Wt(O) = total weight
- Read threshold $T_r(O)$; Write threshold $T_w(O)$
- $T_r(O) + T_w(O) > Wt(O)$: read always reads up-to-date write
- $2 * T_w(O) > Wt(O)$: write always overwrites up-to-date write
- For read: acquire S-lock on read quorum, read all copies & return **highest version number**
- For write: acquire X-lock on write quorum, write all copies & update version number $= max(n) + 1$
- k-tolerant: can tolerate failure of up to k sites
  - Must hold: weights of $n - k$ sites $\geq max\{T_w(O), T_r(O)\}$

Consistency

Consistency Levels

- Eventual Consistency: returns *any value that was written* (might not be valid state)
  - If single variable, always valid state
- Consistent Prefix: returns all writes $\leq k$ (must be valid state)
- Bounded Staleness: returns all writes performed $\leq t_{\text{read}} - T$, where T is time period of staleness, newer writes *may* be visible
- Causal Consistency: $T_1 < T_2$ (T_1 causally precedes T_2) if *any* are true:
  - T_2 is executed after T_1 in **same session**
  - T_2 reads from T_1
  - T_1 & T_2 has same write & T_2 is final write
  - transitivity of causal
- Strong Consistency: returns latest value
- Monotonic Reads: second read returns same OR *more recent*
- Read My Writes

Tradeoffs

- Strength = size of set of allowable results
- Performance = read latency
- Availability = likelihood of success (with server failures)

Pileus

- Lazy Primary-Copy Replication
  - Primary sites store master copies: all updates performed here & ordered by commit timestamps
  - Secondary sites: async propagation
- Distributed SI

Within a session, xacts are **serial**, defines scope for RMW, MR.

* Pileus's eventual = consistent prefix

Pileus Multiversion Storage

- Each **server**:
  - key-range of keys managed by me
  - store = set of (key, value, timestamp)
  - highTS = commit timestamp of **latest** xact by me
  - lowTS = timestamp of latest pruning operation
- Each **primary server** (only this is involved in 2PC):
  - logical clock
  - pending (awaiting commit) = list of (Put-set, proposed timestamps)
  - propagating (after commit) = queue of (Put-set, commit timestamp)
- Pruning Old Versions by increasing lowTS
  - retain all versions with commitTS > lowTS **OR** latest version where commitTS $\leq$ lowTS

Pileus Client

- Each **application client**:
  - each server: key-range, latency, highTS, mapping of real-time to logical time
  - each session: commit timestamps of previous Puts + previous Gets
- `Put(key, value)`: buffered at **client** (visible to Gets in T, but not for ~T)
- Client's `Get(key)`: if rejected: restart xact (if primary), change server (if secondary)
- Server processing of `Get(key, readTS)`:
  1. Accept if readTS $\geq$ lowTS (if primary) OR readTS in [lowTS, highTS] (if secondary)
  2. S updates logical clock (if primary)
  3. S returns (v, v.commitTS, S.highTS) where v is most recent and v.commitTS $\leq$ readTS
- `BeginTx(consistency, key-set)`:
  1. Get MARTS(T) - minimum acceptable read timestamp
  2. (if key-set given) for each k_i in key-set, pick *closest server* that:
     - has key
     - highTS >= MARTS (if primary, ignore this rule)
     - tie break by lower latency, then higher highTS
  3. readTS(T) = min highTS among the servers picked for each key
- `EndTx(consistency, key-set)` for committing (only primaries):
  - Client selects Commit Coordinator (CC), and sends it readTS + Put-set + largest commit timestamp (of all Gets / Puts)
  - CC partitions Put-set for participants + send PrepareCommit with subset
  - CC updates local clock to max(local, largest commit timestamp + 1)
  - On receiving PrepareCommit, each participant:
    - proposedTimestamp = (local clock)++
    - append (local Put-set, proposedTimestamp) to pending list
    - reply with proposedTimestamp
  - CC picks commitTS = max of proposedTimestamp & sends to all
  - On receiving commitTS, each participant:
    - updates local clock = max(local, commitTS + 1)
    - validate commit using SI
    - replies commit / abort
  - if all commit: write commit log & flush (commit timestamp + Put-set), informs client and participants
  - On receiving confirmation, each participant:
    - processes local Put-set + add to propagating (async)
    - when done: inform CC + remove from pending

Blocked by Pending Transaction T'

Block if possible for commitTS(T') $\leq$ commitTS(T)

- `Get(k)` for T: if T' updated same key k & T'.proposedTimestamp $\leq$ readTS(T)
- Validation request for T: both updated same key & T'.proposedTimestamp $\leq$ commitTS(T)
- Replicating update for T: T'.proposedTimestamp $\leq$ commitTS(T)

MARTS(T) - timestamp of version (not xact)

- Strong Consistency (only consider primary servers):
  - let maxTS(k_i) be **max** timestamp among all versions of k_i in primary server of k_i
  - MARTS = **max** of maxTS(k_i) for all k_i in key-set
- Eventual Consistency: 0
- Bounded Consistency: realToLogicalTime(client clock - t)
- Causal Consistency: **max** timestamp of all previous Gets and Puts in current session
- Monotonic Reads: **max** timestamp of **all** previous Gets in current session (key-set doesn't matter)
- Read My Writes: **max** timestamp of **previously committed** Puts in current session for objects accessed by T

Raft Consensus Algorithm

- Use cases:
  - replicate logfile of coordinator in 2PC
  - replicate locks in centralised lock
- Timers
  - Election Timer: follower becomes candidate OR candidate restarts an election if timeout
  - Leader Timer: leader resends RPC to follower if timeout
  - Client Timer: client resends command if timeout
- Election Safety Property: at most one leader per term
  - each server only votes once
  - must get majority votes
- Election Liveness Property: some leader must *eventually* be elected
  - Each server chooses Election Timer *randomly* from [T,2T]
  - Good if T » broadcast time (average RTT)
- Persistent State (on disk) for all servers:
  - currentTerm: latest term server *has seen*
  - votedFor: in current term (can be null)
  - log[]: log entries (index, term, command); 1-indexed
- `RequestVote(candidateId, term, lastLogIndex, lastLogTerm)`, server R replies as follows:
  1. if RPC.term < R.term: reply (R.term, false)
  2. if RPC.term > R.term AND p:
     - R.term = RPC.term; R.votedFor = RPC.candidateId
     - reply (R.term, true)
  3. if RPC.term = R.term AND r.votedFor = NULL AND p:
     - R.votedFor = RPC.candidateId
     - reply (R.term, true)
  4. if RPC.term = RPC.term AND r.votedFor = RPC.candidateId (previous reply lost): reply (R.term, true)
  5. else:
     - if RPC.term > R.term: R.term = RPC.term; R.votedFor = NULL
     - reply (R.term, false)
- p: R is *not* more complete than candidate
  - X is more complete than Y if EITHER:
    - X.lastLogTerm > Y.lastLogTerm
    - OR X.lastLogTerm = Y.lastLogTerm AND X.lastLogIndex > Y.lastLogIndex
- `AppendEntries(leaderId, leaderTerm, leaderCommitIndex, prevLogIndex, prevLogTerm, logs[])`: used for heartbeat too
  1. if leaderTerm < F.term: reply (F.term, false)
  2. if leaderTerm > F.term: F.term = leaderTerm
  3. if logs[] not empty:
     - If preceding don't match: reject & let leader retry with lower index
     - Kill extraneous entries and append
  4. if leaderCommitIndex > F.commitIndex: F.commitIndex = min(leaderCommitIndex, F.lastLogIndex)
  5. reply (F.term, true)
- Leader Completeness Property: if a log entry is committed in a given term, then that entry will be present in the logs of future leaders
- State Machine Safety Property: if a server has applied a log entry, no other server will apply a different log entry for same index

Normal Operations

- Client sends command to leader; Leader appends command to log; Leader sends AppendEntries to all followers
- Once **committed**:
  - Leader executes & returns result to client
  - Leader notifies followers of committed entries in future AppendEntries (then followers will execute it)
- Leader Append-Only Property: leader never overwrites / deletes entries in its log
- Log Matching Property: if two entries in different logs have same index + same term
  - => stores same command
  - => logs in all preceding entries are identical

Committed Log Entries

- **Directly committed** once *leader that created the entry* has replicated it to a majority of servers
  - NOT counted even if same server re-elected
- **Indirectly committed** if preceding a directly committed commit
- Committed entries are guaranteed to be in **ALL** future leaders

Log Inconsistencies

- Missing entries: replication takes time
- Extraneous entries: will be removed by future AppendEntries

Volatile State

- On all servers (init to 0):
  - commitIndex: index of highest log entry *known to be committed*
  - lastApplied: index of highest log entry *applied to state machine*
- On leaders (re-init after election):
  - nextIndex[] for each server (init to leader.lastLogEntry + 1 -> assume all are replicated)
  - matchIndex[] for each server: highest index that's guaranteed to be replicated (init to 0)

Raft Rules

- For all servers:
  - if commitIndex > lastApplied: lastApplied++; apply logs[lastApplied]
  - if request / response contains term > currentTerm: currentTerm = term; convert to follower
- For leaders:
  - Upon election: send initial + periodic heartbeats
  - If received command from client: apply to local log; respond to client **after** applying command
  - if lastLogIndex $\geq$ nextIndex for follower (update follower's state):
    - Send AppendEntries one-by-one; if fail: decrement & retry
  - if knows command has replicated to majority: update commitIndex

Distributed Query Optimisation

Cost Estimation

- $SF(\sigma_{A=v}(R)) \approx \frac{1}{|\pi_A(R)|}$ (uniformity assumption)
- $SF(\sigma_{A<v}(R))$ take ratio
- $SF(\sigma_{p_1 \wedge p_2}(R)) \approx SF(\sigma_{p_1}(R)) * SF(\sigma_{p_2}(R))$ (independence)
- $SF(R \bowtie S) \approx 1/(max(|\pi_A(R)|, |\pi_A(S)|))$ (inclusion)
- $SF(R \ltimes_A S) \approx \frac{|\pi_A(S)|}{|domain(A)|}$ (inclusion)

Communication cost = $T_{\text{fixed overhead per msg}}$ * (number of msgs) + $T_{\text{one data unit}}$ * (total size of data)

Semijoin Optimisation

Eliminate dangling tuples (doesn't join with any tuple)

Assume $|R| < |S|$ and $R \ltimes_A S$

- Cost = $T_{\text{MSG}} + T_{\text{TR}} * |\pi_A(S)|$
- Benefit = $T_{\text{TR}} * |R| * (1 - SF(R \ltimes_A S))$
  - $SF(R \ltimes_A S) = SF_{\text{SJ}}(S.A)$

Semijoin is beneficial if benefit > cost.

- Two query plans are **comparable** if satisfy **ALL**:
  - both plans have *same output schema* (semijoin and join NO!)
  - AND both plans execute *final operator* on *same server*
  - AND both plans *either* unordered *or* sorted in same order
- Plan is better if comparable and lower cost

```
for i := 1 -> n:
 opt({R_i}) = accessPlans(R_i)  # index scan, ...
 prunePlans(opt({R_i}))

for i := 2 -> n:  # for increasing subsets
 for S subseteq {R_1, ..., R_n} st |S| = i:
  opt(S) = None
  for O propersubset S:  # join & SJ all pairs
   for P propersubset O:  # enumerator extension
    other = opt((S - O) union P)  # re-add P
    opt(S) += joinPlans(opt(O), other, O)
    opt(S) += semijoinPlans(opt(O), other, O)
    prunePlans(opt(S))
  ts = 0  # timestamp
  while True:  # fix-point iteration
   S' = new plans with timestamp=ts (prev loop)
   for O subseteq S & O is not None:
    opt(S) += joinPlans(S', opt(O), ts+1)
    opt(S) += semijoinPlans(S', opt(O), ts+1)
    prunePlans(opt(S))
   ts++
   if S' is None: break

return opt({R_1, ..., R_n})
```

- Enumerator Extension: left & right operands may not be disjoint
- Avoiding redundant joins & semijoins
  - Reasonable query plan *heuristic*: for each leaf-to-root path, each predicate only appears once
- Fix-point Iteration: some plans may not be complete
  - Complete = output schema is *same* as not using semijoin
- Vertical Pruning
  - Comparable plans for opt($\{\ldots\}$) can be found outside the entry (because of incomplete plans)
  - Need to prune and compare across entries