CS4224
AY24/25 SEM 1
github/JothamWong

2. Data Partitioning

1. Horizontal Fragmentation (Partition)
   1.1. **Completeness**: $\forall t \in R, \exists R_i \; s.t \; t \in R_i$
   1.2. **Reconstruction**: $R = R_1 \cup \cdots \cup R_n$
   1.3. **Disjointness**: $\forall R_i, R_j (i \neq j \Rightarrow R_i \cap R_j = \emptyset)$

   Techniques:
   1.1. Range Partitioning (on some predicates)
   1.2. Hash Partitioning (Modulo or Consistent Hashing)
      - Consistent Hashing allows even distribution or efficient redistribution of non-uniform data, oblivious to server heterogeneity (with virtual nodes). Key k goes to node N if $h(N-1) < h(k) \leq h(N)$.
   1.3. Derived Horizontal Fragmentation (Based on a **non-nullable foreign key** to another partitioned relation)
2. Vertical Fragmentation
   2.1. Completeness follows. **Reconstruction**: $R = R_1 \bowtie \ldots \bowtie R_n$
   2.2. **Disjointness**: $\forall R_i, R_j (i \neq j \Rightarrow \text{attributes}(R_i) \cap \text{attributes}(R_j) = \{\text{key}(R)\})$
3. Purpose is for co-location of data (geographical)
4. Optimize for access patterns: Avoid **distributed update txns** (update multiple partitions), avoid **scatter-gather** (accesses every partition)

3. Query Processing

Reduction Techniques

Simplify localized query (by eliminating redundant fragments) following conversion of distributed query

1. $R_i = \sigma_{F_i}(R) \wedge \neg(F_i \wedge p) \implies \sigma_p(R_i) = \emptyset$
2. $R_i = \sigma_{F_a \wedge F}(R) \wedge S_j = \sigma_{F'_a \wedge F'}(S) \wedge \neg(F_a \wedge F'_a) \implies R_i \bowtie_a S_j = \emptyset$
3. $S_i = S \ltimes_a R_i$ is a derived horizontal fragment of $R_i \wedge i \neq j \implies S_i \bowtie_a R_j = \emptyset$
4. $R_1, \ldots, R_n$ are vertical fragments of $R \wedge (attr(R_1) - key(R)) \cap L = \emptyset \implies \pi_L(R_1 \bowtie \cdots \bowtie R_n) = \pi_L(R_2 \bowtie \ldots \bowtie R_n)$

Join Strategies for $R \bowtie_a S$

- Case 1: Both R and S partitioned on join key (Collocated)
- Case 2: Only R, not S, is partitioned on join key (Directed or BCast)
- Case 3: Neither R nor S partitioned on join key (Repartitioned or BCast)
- When tackling optimal partitioning, there is no greedy choice. One must enumerate all plans as a suboptimal broadcast now can result in an optimal collocation later.

Join Strategy	Communication Cost (excl. post-join union cost)
Collocated	0
Directed	$size(R)$ if R is being repartitioned
Repartitioned	$size(R) + size(S)$
Broadcast	$(n-1) \times size(R)$ if R is being broadcast

4. Query Optimization

Query plan minimizes (CPU, I/O) cost (max. throughput), or latency

Selectivity Factor

1. **Extended push down of selection over join:** $\sigma_p(R \bowtie_{p'} S) = \sigma_{PR}(R) \bowtie_{p'} \sigma_{PS}(S)$ iff $p = PR \wedge PS, attr(PR) \subseteq attr(R) \wedge attr(PS) \subseteq attr(S)$
2. Joins can be distributed over union: $(E_1 \cup E_2) \bowtie (C_1 \cup C_2) = (E_1 \bowtie C_1) \cup (E_1 \bowtie C_2) \cup (E_2 \bowtie C_1) \cup (E_2 \bowtie C_2)$
3. - Uniformity assumption: uniform distribution of values in attr
   - Independence assumption: attrs are independent
   - Inclusion assumption: For $R \bowtie_A S$, if $\|\pi_A(R)\| \leq \|\pi_A(S)\|$ then $\pi_A(R) \subseteq \pi_A(S)$
4. $SF(\sigma_{A=v}(R)) \approx \frac{1}{\|\pi_A(R\|}$
5. $SF(\sigma_{A<v}(R)) \approx \frac{v - \min(\pi_A(R))}{\max(\pi_A(R)) - \min(\pi_A(R)) + 1}$
6. $SF(\sigma_{p_1 \wedge p_2}(R)) \approx SF(\sigma_{p_1}(R)) \times SF(\sigma_{p_2}(R))$
7. **Join selectivity**: $SF(R \bowtie_A S) \approx \frac{1}{max(\|\pi_A(R)\|, \|\pi_A(S)\|)}$
   - Inclusion assumption: Every R tuple participates in join.
   - Uniformity assumption: Every R tuple joins with $\frac{\|S\|}{\|\pi_A(S)\|}$ tuples.
8. **Semi-Join selectivity**: $SF(R \ltimes_A S) \approx \frac{\|\pi_A(S)\|}{\|domain(A)\|}$

Cost estimation

1. CPU/IO Cost = $T_{\text{cpu/io}} \times$ #cpu/io insns
2. Comm Cost = $T_{\text{MSG}} \times$ #messages + $T_{\text{TR}} \times$ size of data
   - T_{MSG} = fixed overhead for each message transmission
   - T_{TR} = time to transmit one data unit

Optimization w/ Semi-Joins

$R \bowtie_A S = (R \bowtie_A \pi_A(S)) \bowtie_A S$

- $R \ltimes_A S$ or $R \bowtie_A \pi_A(S)$ eliminates dangling tuples in R wrt $R \bowtie_A S$.
- $R \ltimes_A S$ is beneficial iff $Benefit = T_{TR} \times size(R) \times (1 - SF(R \ltimes_A S) > Cost = T_{MSG} + T_{TR} \times size(\pi_A(S))$

Group-By Optimization

Refer to exam attachment.
For $G_{A,F}(R)$, $A \subseteq attrs(R)$ and $F = \{T_1 = f_1(e_1), \ldots, T_n = f_n(e_n)\}$, where each A_i is a grouping column, each f_i is an aggregate function, each e_i is an expression of attributes in R and each T_i is a column alias.
$alias(F) = \{T_1, \ldots, T_n\}$

5. Storage

Log-Structured Merge Table

- MemTable + SSTables + commit log
- Deleted records marked with tombstones ($\perp$)
- Flush to new SSTable when MemTable is full
- SSTables are sorted by and associated with range of key values and creation timestamp

LSM Compaction

1. Remove stale (older) values. Keep tombstones if newer.
2. Let r be a record in R and let V_r be the set of all versions of $r \in D$
3. **Size-tiered Compaction Strategy**
   - Each tier has approximately the same size, and a higher tier is larger than the previous tier.
   - Compaction is triggered when the number of SSTables at a tier L reaches a threshold, then all SSTables in Tier L merged into a single SSTable in Tier L+1. Tier L is empty and Tier L + 1 can either be +1 (most cases) or -1 if last level and all are tombstones.
   - **Search:** Top-down from MemTable, Tier 0 tables to Tier m tables, most recent timestamp first.
   - There is 1 unique version of r in each table in each tier. So if you have $N_0 + N_1 + N_2$ tables, (N_0 = number of tables at tier 0), you have $N_0 + N_1 + N_2$ versions of r in total.
4. **Leveled Compaction Strategy**
   - After merge, split into properly-sized tables.
   - SSTables at level 0 may overlap
   - $L \geq 1$: SSTables at the same level do not overlap, have the same size
   - SSTable at level L overlaps with at most F SSTables in level L+1.
   - If the max num records in R is n and the size of each record is m MB, the max size of R is mn. Let L denote num levels to store R. In worst case, last level of LSM stores a version of each record in R. Therefore $F^{L-1} < mn \leq F^L \rightarrow log_F(mn) \leq L < log_F(mn) + 1 \rightarrow L = ceil(log_F(mn))$
   - Increasing F reduces number of levels of LSM which improves worst case I/O for searching. But larger F means more overlapping tables to be merged during compaction, so it increases I/O cost of compaction.
   - In LCS, it is possible for 2 tables in L (S1, S2) to overlap with the same table (S3) in L+1. Denote the intersection of S2 and S3 as O3. S2 overlaps with S3 and F-1 other tables in L+1. When compacting S1, if the records in O3 are distributed across two tables (as table is full), S2 will violate invariant. The fix is to identify O3, and if the entire O3 cannot fit into the current new table, move them all to a new table.
   - **Compacting**
     - $L = 0$: **All** SSTables at level 0 are merged with all overlapping SSTables at L1
     - $L \geq 1$: Let v be the ending key of the last compaction. Next SSTable S to compact is first SSTable that starts after v if it exists, otherwise go to smallest start value. Merge S with all overlapping SSTables at level L+1.
   - **Search:** Top-down from MemTable, Tier 0 tables to Tier m tables, most recent timestamp first. **Check if key falls in table's range before searching within table.**
   - For V_r, there is 1 version of each table in Tier 0 and only 1 unique version in Tiers 1 and above. So you have $N_0 + k$ where k is number of Tier 1 and above SSTables.

LSM Search Optimization

Each SSTable file consists of a sequence of data blocks.

1. **Sparse Index**: To find the block, build a sparse index of "first key value in i^{th} block" $\rightarrow$ "address of i^{th} block". Search within block of covering range.
2. **Bloom Filter**: To test if a key x exists in a block, build a bloom filter B for each block S. If $\exists i \in [1,k] \; s.t. \; h_i(x) = j$ and $B[j] = 0, x \notin S$.

Indexing

- **Local Index**: An index for each partition is built.
- This is bad for search: as you need to search the index of each partition
- Better for updates, as you only need to update the index for each partition
- **Global Index**: The index is a derived partition from the data partition.
- Bad for updates, as the index may not be on the same fragment as the data.
- Good for searching the index, as you only need to search one index server. But bcos data can be on different shards, still may end up querying multiple shards.

6. Commit Protocols

- **Log** Sequential file of records in **non-volatile/stable storage** (multiple copies)
- **Recovery manager**: Supports Abort, Commit and Restart to preserve atomicity and durability of **local** txns.
  - **Abort: Write-ahead logging (WAL) protocol**
    - Uncommitted update to DB not flushed til log with before-image is flushed.
    - Restore log record's before-image in reverse order.
  - **Commit: Force-at-commit protocol**
    - Do not commit a Xact until after-images of all its updated records are in stable storage (DB or log). Enforced by writing a **commit log record** r for Xact and flushing all log records (up to and including r) for Xact to disk.
    - An Xact is considered committed if its commit log is written to stable storage
  - **Restart: Redo + Undo**
    1. **Redo Phase:** Scans log records in forward direction to redo updates, keeping track of active txns.
    2. **Undo Phase:** Abort active txns.
- **Commit Protocol:**
  - Txn coordinator (TC) coordinates with Txn Managers (TMs) to execute txn T at multiple sites, and ensures **atomicity** of distributed txn.
  - **Log Records:** Log writes are **forced** or synchronous if it must be flushed to disk before the next message can be sent, else it is not forced/asynchronous.
  - **Site failures:**
    - Detected by **timeouts**, which invoke **termination protocol** at operational sites. Termination protocol is **non-blocking** if it permits a transaction to commit/abort at sites w/o waiting for recovery.
    - On restart, invoke **recovery protocol** at failed site. Recovery protocol is **independent** if no communication with another site is necessary to determine how to terminate a txn.

2PC Protocol

Two-Phase commit
1. Voting phase: TC collects votes from TMs
2. Decision phase: TC broadcasts global decision to TMs.

2PC is synchronous within one state transition.
See exam attachment for state transitions, recovery and termination protocols.
2PC Cooperative Termination Protocol
Reduce probability of blocking by failure in coordinator.
- TC includes addresses of all participants in "Prepare" msg.
- When a participant P times out in READY state, bcast "Decision-request" msg.
- When another participant receives it, respond as follows:

State	Actions
INITIAL	Replies "Abort"; Unilaterally aborts
READY	Replies "Uncertain"
COMMIT	Replies "Commit"
ABORT	Replies "Abort"

- P terminates txn with decision, if any, and sends it to all participats that replied "Uncertain". Else P remains blocked.

Other blocking scenarios
- C fails after sending Global-commit, P_1 fails after recieving Global-commit, others are in READY state.
- C fails before recieving any vote, P_1 fails after voting, others are in READY state.

3PC-1 Protocol

Three-Phase commit. Non-blocking in absence of comm. or total site failure. May block in event of total site failure, but correctness is guaranteed.
1. Voting phase: TC collects votes from TMs
2. TC disseminates voting outcome to TMs if there is no abort vote.
3. Decision phase: TC broadcasts global decision to TMs.

See exam attachment for state transitions, recovery and termination protocols.
3PC Termination Protocol 1
1. Elect a new coordinator C'.
2. C' sends "State-request" msg and obtains current states of participants.
3. C' terminates txn as follows:
   3.1. If there is some TM in COMMIT,
      i. C' sends "Global-commit"
   3.2. else if no Tm is in PRECOMMIT,
      i. C' sends "Global-abort" to all
   3.3. else, C' sends "Prepare-to-commit" to TMs in READY state. After recieving "Ready-to-commit" from them, send "Global-commit" to all.

- If any TM times out, elect a new TC.
- If any participant fails, TC will ignore it.
- If a participant recovers during the termination protocol, it's blocked until after the protocol finishes.

Handling Total site failure
Recovering TMs remain blocked until a TM P recovers:
- **Case 1:** P recovers independently (it is READY/ABORT/COMMIT). P will notify recovered TMs of global decision.
- **Case 2:** P was the last TM to fail
  P terminates the txns by executing Termination Protocol 1 among recoverd TMs.

3PC-2 Protocol

Goal: Under comm. failure, ensure correctness (consistent decision) made by **multiple coordinators**.
3PC Termination Protocol 2
1. Elect a new coordinator C'.
2. C' sends "State-request" msg and obtains current states of participants.
3. C' terminates txn as follows:
   3.1. If there is some TM in COMMIT,
      i. C' sends "Global-commit"
   3.2. else if there is some TM in ABORT,
      i. C' sends "Global-abort"
   3.3. else if there's at least 1 PRECOMMIT, no COMMIT/ABORT & majority READY+PRECOMMIT states,
      i. C' sends "Prepare-to-commit" to those not in PRECOMMIT.
      ii. Those recieving change to PRECOMMIT and reply "Ready-to-commit".
      iii. If "Ready-to-commit" msg + PRECOMMIT is majority, C' sends "Global-commit"; else, all **block**.
   3.4. else if there's no COMMIT nor ABORT & majority INITIAL+READY+PREABORT states,
      i. C' sends "Prepare-to-abort" to those not in PREABORT.
      ii. Those recieving change to PREABORT and reply "Ready-to-abort".
      iii. If "Ready-to-abort" msg + PREABORT is majority, C' sends "Global-abort"; else, all **block**.
   3.5. else, all **block**.

Blocked TMs periodically execute the above protocol, as do failed TMs upon recovery.
3PC-2 is non-blocking in absence of comm. failure, so long as majority of TMs are operational.
Reasoning about whether an operation is forced or not forced, think about what happens if it crashes before it writes the op, does that contradict the global decision?
3PC-1 can global abort even if one P is in pre-commit. C crashes immediately after. Rest of P are in READY and elect new C. Termination protocol.

7. Concurrency Control

Revision

- S and S' are **View equiv.** if
  1. Every T_i that reads A from T_j in S also reads A from T_j in S'.
  2. For each A, the same txn performs the final write in S' as in S.
- Pairs of accesses conflict if they are on the same obj., are from different txns, and one of them is a write.
- S and S' are **Conflict equiv.** if they preserve the order of all pairs of conflicting accesses in committed txns.
- S is **VSS/CSS** if S and some serial (non-interleaved) sched. are **view/conflict equiv**.
- VSSs with no blind writes are also CSSs.
- If in S, every T commits after all T's it reads from, S is a **recoverable sched.**
- Assume blocked lock reqs. are FIFO queued (no starvation), and checked on each release.
- Terminating a txn removes all its locks from req. queue.
- Writes of Ts in **strict scheds.** are not read from or overwritten by another txn until T terminates. Thus they can be recovered with before-images (in logs).
- **2PL:** Uses S and X-locks, cannot request more locks after releasing first lock.
- **S2PL:** Can only release locks when terminating.
- **MVCC:** Read-only txns never block, become blocked, or become aborted.
- S and S' are **MV view equiv.** if every T_i that reads A_k from T_j in S also reads A_k from T_j in S'.
- Each read in a **Monoversion scheds.** returns most recent-version write.
- S is **MVSS** if S and some serial **monoversion** sched. are **MV view equiv**.
- S2PL-S $\subset$ 2PL-S $\subset$ CSS $\subset$ VSS $\subset$ MVSS
- **SI:** $R_i(O)$ reads the latest write in T_i, if any. Else, they read from the latest $W_j(O)$ where T_j has the largest commit timestamp smaller than T_i's start timestamp.
  1. **FCW:** In pairs of concurrent txns, the first txn to commit causes the other to abort.

2. **FUW:** Txns requesting and blocked on an X-lock abort upon commit of the blocker txn. Txns upon obtaining an X-lock abort if they detect it has been updated by a concurrent txn.

- SI protocols can produce non-serializable schedules (Write Skew/Read-Only Txn anomaly).
- To disprove SI, check that any concurrent updates must be disjoint. First scan all schedules and find two Txns that write to the same object. Say W4(c), W2(c). Then, you must show that C4 happens before the first operation in T2 or find a counter example by seeing if you can find a chain to any operation before W4(c) in the same local txn.
- To disprove S2PL, check that a txn must release locks.

Distributed CC

- Let $T = \{T_1, ... T_n\}$ be a set of distributed txns executed over m sites with local schedules $\{S_1, ... S_m\}$.
- A schedule S is a **global schedule** for T and the local schedules if each S_i is a subsequence of S.
- A **serializable global schedule** S is **view/conflict equiv.** to some serial schedule S' over T. To approach this, for each local schedule S_i, derive a possible serial schedule for each local schedule. If the union of all the serial scheds. are compatible (acyclic) a global serial schedule exists.
- **C2PL:** Central site manages all lock requests.
  Deadlock Detection: Each site maintains a local **Wait-For Graph (WFG)**, periodically transmitted to central site to build global **WFG**.
- **D2PL:** Each lock is held at the same site as the obj.
- **CSI:** CC site (different from TM_A) assigns start and commit timestamps. Queries from TC to obtain start timestamp also returns latest prior commit timestamp.
  Uses **FUW**, where X-locks are managed locally by each TM_A.
  1. To read x at Site A, TC sends read req and last commit T_i to TM_A. TM_A sends most recent version of x wrt lastcommit timestamp.
  2. To write x at site A, send write req to TM_A. Check if X-lock can be given. if granted, TM_A updates X and sends notif to TC, otherwise T_i is blocked.

  When T commits and releases locks, all T_i blocked by T abort.
  On commit, execute **2PC variant:**
  1. TC includes start and commit timestamps of T_i in PREPARE msg.
  2. On receipt, participants (with obj X-locks held by T_i) check for WW-conflicts between T_i and committed concurrent txns (i.e. if version number of obj is between PREPARE start and commit timestamps).

8. Replication

- **One-copy database** = non-replicated database
- **Mutually consistent** = all replicas of data items have identical values.
  Strong: identical at the end of each update txn; **Weak:** eventual consistency (end of sched).
- **Replicated data (RD)**: schedules on replicated database. Vice versa for **1-copy** schedule.
- T_j reads x from T_i in RD if
  1. for some copy x_A of x, $W_i(x_A)$ precedes $R_j(x_A)$ and
  2. there is no $W_k(x_A), k \neq i$ that occurs between $W_i(x_A)$ and $R_j(x_A)$
- S_{RD} is **1SR** if it is equivalent to a serial one-copy schedule S_{1C}, where equivalence is
  1. T_j reads x from T_i in S_{RD} iff T_j reads x from T_i in S_{1C} and
  2. for each final write $W_i(x)$ in S_{1C}, $W_i(x_A)$ is a final write in S_{RD} for **some copy** x_A
- To check **1SR**, look at the reads-from relationship for each local schedule and check the precedence graph in the serial schedule for cycles. Then check if final write is satisfied for each serial final write.
- Replication protocols are defined as WHEN+WHERE.
- Eager (synchronous update): Propagates updates to all replicas affected by xact before commit. Enforces strong mutual consistency, ROWA.
- Lazy (async): Xact updates only one replica, updates may propagate later (refresh txn).
  - Need to preserver order of different refresh txns (as well as writes within txn) across all sites.
  - Use the commit timestamp of original txn.
  - Sites receiving refresh txn need to grant X-locks.
- Centralized: Update is applied to master copy and propagated to slave copies.
- Distributed: Update can be applied to any copy and propagated to other copies.
- To distinguish centralized vs distributed, check if T write to another site when it has its own local copy.
- Assume S2PL, statement-based replication.
- Under lazy distributed, Last-Writer-Wins heuristic (timestamp-order) needed to reconcile conflicting concurrent updates by multiple xacts at different sites/copies. i.e. refresh updates with an older txn timestamp than local copy gets ignored. Heuristic can cause non-blind writes (RW-updates) to be lost.

Eager is always 1 copy serializable Lazy is not guaranteed to be 1 copy serializable, even when just 1 txn! Lazy single-master: read local copy, which can be stale cos master update is async Lazy distributed: no 1SR and inconsistent update: multiple xacts can update different copies of same data concurrently at different sites, requires last writer updates (only works for blind writes)

Handling Failures

Single-master replication, with timeout-based detection.

- **Failure of slave site:**
  1. Lazy replication: Sync unavailable ones later when they become available.
  2. Eager replication: ROWAA. Update available replicas and terminate txn, sync unavailable replicas later
- **Failure of master:**
  CAP Theorem: In a **P**artitioned network,
  1. Forfeit **A**vailability: Wait (block) for master site/network recovery, or
  2. Forfeit **C**onsistency: Elect new master. Need to ensure at most one partition can have an operational master; else inconsistency.

Quorum Consensus Protocol

- Assign non-negative weights and version nos. to all copies of an object O, which sum to some $Wt(O)$.
- For k-tolerant, $(n-k)w \geq max\{T_w(O), T_r(O)\}$
- **To Read:**
  1. Acquire S-locks on a quorum of copies whose sum-of-weights exceed $T_r(O)$
  2. Return copy within quorum with most recent version no.
- **To Write:**
  1. Acquire X-locks on a quorum of copies whose sum-of-weights exceed $T_w(O)$
  2. Get max version no. within quorum, n.
  3. Write to all copies in quorum, setting version no. to $n+1$.
- Note that $T_r(O) + T_w(O) > Wt(O)$ and $2 \times T_w(O) > Wt(O)$

9. Consistency

Pileus

1. Range-partitions and replicates with lazy centralized: primary and secondary sites. Updates are ordered by commit timestamps and performed at primary sites. Updates are received in timestamp order at secondary sites.
2. Concurrency control uses distributed snapshot isolation: prevent concurrent updates, readTS(t), commitTS(t)
3. Server maintains the following information:
   3.1. **key-range**: range of keys maintained by server
   3.2. **store**: set of (key, value, timestamp)
   3.3. **highTS**: commit timestamp of latest txn processed by server
   3.4. **lowTS**: timestamp of server's most recent pruning operation
4. Each primary server additionally maintains
   4.1. **logical clock** for assigning commit timestamps
   4.2. **pending** = list of (Put-set, proposed timestamp) pairs for uncommitted txns
   4.3. **propagating** = queue of (Put-set, commit timestamp) to be async-sent to secondary replicas
5. **Pruning old data**: For each object O at server S, all versions of O with commitTS ≤ S.lowTS are pruned, **except for one (w/ latest version)**
6. BeginTx(Consistency level L, key-set KS)
   6.1. Determine **readTS(T)** for new Xact T based off L, it determines the snapshot that T accesses for all Get operations.
   6.2. If Get(k) is serviced by server S, S will return the latest version v of key k at S s.t v.commitTS $\leq$ readTS(T)
   6.3. First, compute **MARTS**: Minimum Acceptable Read Timestamp. As long as readTS(T) $\leq$ MARTS(T), L is guaranteed
   6.4. A server S is a **candidate server** for Get(k) if
      i. S contains k in its key-range AND
      ii. Either S is a primary server for k or highTS[S] $\geq$ MARTS(T)
   6.5. Among the candidate servers for Get(k), pick S that minimizes latencies, if tie, maximize highTS[S].
   6.6. For each $k_i \in KS$, let S_i denote server selected for Get(k)
   6.7. readTS(T) $= \min\{$highTS[S]$| k_i \in KS\}$
7. Get(key)
   7.1. Becomes Get(key, readTS(T)) to S which is processed as
   7.2. If S is the primary server for key, S accepts request if readTS(T) $\geq$ S.lowTS, and S updates its logical clock to max of local clock or readTS(T)
   7.3. If S is a secondary server for key, S accepts request if readTS[T] $\in$ [S.lowTS, S.highTS]
   7.4. If S accepts the request, then S returns (v, v.commitTS, S.highTS) where v is the most recent version of key in S with v.commitTS $\leq$ readTS(T). Otherwise S rejects the Get request
8. **MARTS**: Minimum Acceptable Read Timestamp
9. **Strong**: Contains results of all Xacts that committed before start of T. Let maxTS(k_i) be max ts among all versions of key k_i in the primary server for k_i. MARTS(T) $= \max\{$maxTS$(k_i) | k_i \in KS\}$
10. **Eventual**: In Pileus, this is equivalent to consistent prefix consistency (all writes up till and including the k-th write). In non-pileus, it is an arbitrary subset of the superset of all tables. MARTS $= 0$.
11. **Read-my-writes**: MARTS(T) = max ts of all previously committed Puts for keys accessed by T in current session
12. **Monotonic Reads**: MARTS(T) = max ts of all previous Gets (**any key**, including those not in KS in BeginTx!!!!!!!!!!!!).
13. **Bounded(t)**: Snapshot contains results of all Xacts committed from start t - deltaT to start of t. MARTS(T) = **realTimeToLogicalTime(client's clock time - t)**
14. **Causal Consistency**: Snapshot contains results of all Xacts that **causally precede** T. If $T_1 < T_2$ then commitTS(T_1) < commitTS(T_2). MARTS(T) = max ts of all previous Gets and Puts for any key (!!!) in current session
15. **T1 causally precedes T2** if any of the following hold
    15.1. T_2 is executed after T_1 in the same session
    15.2. T_2 reads some object written by T_1
    15.3. T_1 and T_2 both performed a put on the same object and T_2 commits after T_1
    15.4. There is some Xact T_3 where $T_1 < T_3$ and $T_3 < T_2$
16. EndTX
    16.1. Only primary servers with data updated by T will be participants in the 2PC process, **commit coordinator** (CC) is among them
    16.2. Client sends a **commit request** containing: **readTS(T)**, set of Puts for T (**Put-set**), largest commit timestamp among all Gets/Puts in the session (**LCT** to enforce read-my-writes and causal consistency).
    16.3. CC partitions Put-Set into $PS_1 \cup \cdots \cup PS_n$ where P_i is primary server for keys in PS_i.
    16.4. When CC receives commit request from client, CC updates local clock to max(local clock timestamp, LCT+1). Then sends **prepare-commit** request to each P_i alongside PS_i
    16.5. When P_i rcvs **prepare-commit** from CC, P_i sets proposedTimestamp = local clock ts, increments local clock TS, appends PS_i to pending list (list of uncommitted txns) and replies to CC
    16.6. From all proposedTS, CC selects max as **commitTS(T)** and sends **commitTS(T)** to all participants
    16.7. Upon receiving **commitTS(T)** from CC, P_i updates local clock to max(local clock timestamp, commit(TS)+1), validates whether it can commit T (First updater win?), and sends abort or commit reply
    16.8. if all P voted to commit, CC commits T by writing a **commit log record** to stable storage (contains commit ts and put-set of T), inform client that T has committed and inform participants
    16.9. When P_i rcv commit decision, P_i processes PS_i by creating new object versions using **commitTS(T)**, appends PS_i to **propagating queue** (for lazy replication). When P_i has processed PS_i for T, P notifies CC that P has completed T and removes T from pending. Then async updates PS_i.

10. Raft Consensus

Server State

Persistent state (survives crashes):
1. **currentTerm**: Latest term server has seen (initialized to 0)
2. **votedFor**: CandidateId that received vote in currentTerm (null if none)
3. **log[]**: Log entries, each containing (index,term,command), first index 1

Volatile state (all servers, initialized to 0):
1. **commitIndex**: Highest log entry known to be committed
2. **lastApplied**: Highest log entry applied to state machine

Leader-only volatile state (reinitialized after election):
1. **nextIndex[i]**: For each server, index of next log entry to send
   - Initialized to (leader's last log index + 1)
   - Used to quickly restore log consistency after failures
2. **matchIndex[i]**: For each server, highest replicated log entry
   - Initialized to 0, increases monotonically
   - Used to track commitment progress

RPCs & Timers

RequestVote RPC:
1. **Arguments**:
   - candidateId: Candidate requesting vote
   - term: Candidate's term
   - lastLogIndex: Index of candidate's last log entry
   - lastLogTerm: Term of candidate's last log entry
2. **Response**: (term, voteGranted)
3. *Vote granted when*:
   - Candidate's term $\geq$ current term (current term will be updated)
   - Server hasn't voted for different candidate
   - Candidate's log is at least as complete as receiver's
4. **If RPC.term > current term** and did not vote, R.votedFor=null
5. **If already voted (R.term=currentTerm) and same cand**, return True!

AppendEntries RPC:
1. **Arguments**:
   - leaderId: Current leader's identifier
   - term: Leader's term
   - leaderCommit: Leader's commitIndex
   - prevLogIndex: Index of log entry immediately before new ones
   - prevLogTerm: Term of prevLogIndex entry
   - entries[]: Log entries to store (empty for heartbeat)
2. **Response**: (term,success), term=current term of follower F, success=True if F contain entry matching prevLogIndex n prevLogTerm, false otherwise
3. *Sending by Leader*:
   - Periodically send as heartbeat. entries is empty in this case.
   - Send log entries starting at nextIndex[F] index. Update nextIndex[F] & matchIndex[F] if success, decrement nextIndex[F] and retry if fail.
   - Update commitIndex to N, s.t. $N > commitIndex$, a majority of $matchIndex[i] \geq N$ and $log[N].term$=currentTerm if N exists
4. *Processing by followers*:
   - Reject if leader's term < currentTerm. Else update currentTerm if higher (reflected in response)
   - Reject if log doesn't contain entry at prevLogIndex with prevLogTerm
   - If conflict found, delete existing entry and all that follow
   - Append any new entries not already in log
   - Update commitIndex if leader's is higher (commitIndex=$\min$(leaderCommit,index of last entry in F's log)

Timer System:
1. **Election timer**: Random timeout in [T,2T]
   - Reset on valid RPCs from current leader
   - Triggers new election when expired
2. **Leader timer**: Periodic trigger for heartbeats/replication
3. **Client timer**: Command retry mechanism

Leader Election

Election Protocol:
1. Follower increments term and becomes candidate when election timer expires
2. Candidate votes for self and sends RequestVote RPCs
3. Upon receiving majority votes, becomes leader and sends heartbeat
4. Reverts to follower if discovers current leader or higher term
5. Starts new election if timer expires before conclusion

Liveness Properties:
1. Random timeouts prevent simultaneous candidates
2. Election timeout $\gg$ broadcast RTT ensures stability
3. Term mechanism breaks deadlocks

Log Replication

X's log is more **complete** than Y if
1. X's last log term is greater than Y's, or
2. The terms are equal, and X's last log index is greater

Normal Operation: Leader appends locally and broadcasts via AppendEntries.
Commitment Rules: A log entry can only be committed through one of two paths, with a crucial **term-specific constraint**:
1. **Direct Commitment**: An entry becomes directly committed only when the leader that created the entry in the current term successfully replicates it to a majority of servers within that same term. **Simply having an entry appear in a majority of logs is insufficient - the replication must happen by the leader that created it, within its own term.**
2. **Indirect Commitment**: When an entry achieves direct commitment, all previous entries in the log automatically become indirectly committed, regardless of their terms. Ensures no gaps in commitment.

The term-specific requirement for direct commitment prevents scenarios where entries from previous terms that achieved majority replication but weren't fully committed could conflict with newer entries. Once committed through either path, a leader executes the command, responds to client, and notifies followers via AppendEntries. Followers then execute the committed commands in order (increment **lastApplied** and apply log[lastApplied] to state machine)
Client Interaction: Clients contact random server initially and get redirected to leader if needed. Each command includes unique serial number to prevent duplicates, used by leaders to cache responses. If timeout, reidentify leader and resend.
Read Operations: Standard approach logs reads as normal entries. Optimized approach allows direct reads after leader commits no-op entry at term start, confirms leadership via heartbeat, and verifies state machine currency.
Consistency Guarantees: Same index and term imply identical history (preceding entries). Leaders append only, never overwrite. Committed entries persist in all future leader logs, with no conflicts possible at same index.

Example Scenario

Consider a 5-server cluster with logs:

```
Term:   1  2  3  4  5  6  7
S1:     1  1  2  2  2  2  4
S2:     1  1  1  3
S3:     1  1  2  2  2
S4:     1
S5:     1  1  2  2  2  2
```

Key Observations:
1. Entry (1,1) appears in all logs: guaranteed preserved

2. Only S2/S5 qualified as leaders if S1 fails:
   - S4: Log too short
   - S3: Missing entries present in S5
   - S2: Can overwrite later entries with term 3
   - S5: Most complete log matching old leader
3. Committed entries analysis:
   - (1,1): Never overwritten (universal)
   - (2,1): Required for majority consensus
   - (3,2)-(5,2): Could be overwritten despite majority