CS4224
Bennett Clement

2 Partitioning

Pros	Cons
Application's locality of access support	Overfragmented relation
Scale to manage large data / workload	More complex integrity constraint checking
Performance improvement with parallelized query execution	

Strategies

1. Horizontal fragmentation (a.k.a. sharding)
2. Vertical fragmentation
3. Hybrid fragmentation

Desirable properties of fragmentation

1. Completeness: Each item in R can also be found in one of its fragments
2. Reconstruction: R can be reconstructed from its fragments
3. Disjointness: Data items are not replicated

Sharding

Horizontal Fragmentation. Use filters to partition.

Techniques

1. Range partitioning: using predicates on some attributes of R
2. Hash partitioning:
   1. Modulo Hashing: put in server $h(a)$
   2. Consistent Hashing (with virtual nodes to account for differences in server's performance and to better distribute data load)
3. Primary Horizontal Fragmentation: fragmenting based on the queries, all attributes are of the relation.
4. Derived Horizontal Fragmentation: Partition a relation R based on the partitioning for a related relation S.
   - To be complete need $R.A \subseteq S.A$
   - To be disjoint: $S.A$ must be a key.

Complete Partitioning wrt Query

Let $F = \{R_1, \ldots, R_m\}$ be a partitioning of relation R, and let Q be a query on R, we say F is a complete partitioning of R wrt Q if for every fragment $R_i \in F$, either every tuple in R_i

matches Q or every tuple in R_i does not match Q.

Minterm Predicate Partitioning

Let P be the predicates of the set of queries Q.

A **minterm predicate** m for P is the conjunction of all the predicates in P of the form $m = p_1^* \wedge p_2^* \dots p_n^*$ where each p_i^* is either p_i or $\bar{p_i}$.

Let MTPred(P) denote the set of all miterm predicates for a set of predicates P.

Theorem: If F is a minterm predicate partitioning of R that is equivalent to MTPred(P), it is also a complete partitioning wrt every query in Q.

Vertical Fragmentation

Use projection to split columns into different fragments.

Can use attribute affinity measure $\text{aff}(A_i, A_j)$ to measure how often attributes are referenced together in queries, and apply clustering algorithm on $\text{aff}(A_i, A_j)$

3 Storage

LSM (Log-Structured Merge) Storage is the most common used storage for distributed db. It improves write throughput by "converting" random I/O to sequential I/O (leveraging append-only updates instead of in-place updates)

Components

LSM Storage for a relation R(K,V) consists of :

1. A main-memory structure **MemTable**
   - contains the most recent updates
   - updated in-place (deleting records = marking it as tombstones)
   - When the size of MemTable reaches a certain threshold, the records in MemTable are *sorted* and flushed to disk as a new SSTable.
2. A set of disk-based structures **SSTables** (Sorted String Tables)
   - immutable structures, sorted by relation's key K
   - each SSTable is associated with a range of key values and a timestamp (for ordering of SSTable)
3. A **commit log** file
   - each new update is appended to commit log & updated to MemTable

Overtime, the SSTable records might be fragmented, and there might be a lot of tombstones & stale values (as we always append updates, and not update in-place).

Compaction

STCS (Size-tiered Compaction Strategy)

- SSTs are organized into tiers with SSTs in each tier having approximately the same size (larger as higher tier number).
- The SST in Tier 0 all have same size, since they all come from MemTable
- Compaction is done by merging all SSTs in tier L to a single SST in tier $L+1$ using a k-way merge
- Compaction is triggered at tier L when the number of SSTs reaches a threshold

Cons: overlapping key ranges makes it slower to search as we potentially need to look at all SSTs.

LCS (Leveled Compaction Strategy)

- SSTs are organized into a sequence of levels
- For each level $L \geq 1$
  - Each SST has the same size and do not have overlapping key ranges
  - Each SST at level L overlaps with at most F (compaction factor) SSTs at level $L+1$
- Compaction is done by:
  - $L \geq 1$:
    - Select an SST S at level L that starts after (wrap-around) the ending key of the last compacted table at level L.
    - Merge S with all overlapping SSTs at level $L+1$
    - Pack as many KVs into a SST except if the current SST is full or adding another KV will violate the compaction factor F property.
  - $L = 0$: Merge all SSTs at level 0 with all overlapping SST at level 1.
- Compaction is triggered when:
  - $L \geq 1$: When $Size(L)$ (total size of all level-L SSTs) $> F^L$
  - $L = 0$: the number of level-0 SST reaches a threshold (e.g. 4)

Searching

Searching is done from the MemTable, level-0, level-1, ... Each level goes from the latest timestamp to the earliest.

Each SST is stored as a file consisting of a sequence of data blocks. To optimize search we can do two things:

Sparse Index

How to quickly locate SST block for a given search key?

We can build a sparse index $[k_1, k_2, \cdots, k_n]$ for each SST where k_i is the first key value in the i-th block of the SST.

Bloom Filter

How to quickly determine whether a search key exists in a SST block?

We can build a bloom filter for each block. Note that bloom filter might have false positives but cannot have false negatives. Also, bloom filter is space efficient.

Indexing

1. Local indexing:
    - Build an index based on the local data.
    - Updates are cheap, but search might need to hit all nodes (expensive)
2. Global indexing:
    - Build an index based on global data (not related to local data in the node)
    - Updates are expensive, but search might be cheaper (depends on the number of nodes accessed to get the data requested)

4 Distributed Commit

Review:

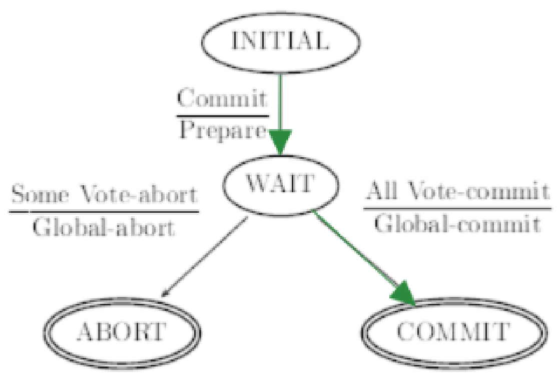
1. Recovery (undo, redo, commit, abort, restart: redo + undo)
2. WAL: don't flush an uncommitted update until the log record containing its before-image has been flushed to the log
3. Force-at-commit: Commit a xact only after the after-iamges of all its updated pages are in stable storage (log or disk)

Define

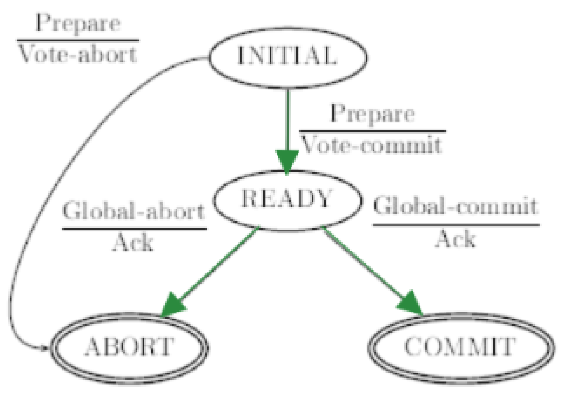
- Originating site: site where Xact is initiated
- Xact coordinator (TC): transaction manager (TM) at originating site
- Failures in DDBMS
    - Site failures: fail stop model (a site is either working correctly or not working at all)
    - Communication failures: lost messages / network partitioning
- Log Records:
    - Forced: if log record must be flushed to disk before sending msg
    - Non-Forced: asynchronous write to log
- Termination protocol: How operational sites deal with failures (detected with a timeout)?
    - non-blocking property: permits a transaction to terminate at operational sites without waiting for recovery of the failed site
- Recovery protocol: How a failed site recovers after being restarted?
    - independent property: can determine how to terminate previously running xacts without having to consult any other site

Two-Phase Commit (2PC)

1. Voting Phase: PREPARE + Vote-Commit/Vote-Abort
2. Decision Phase: Global-Commit/Global-Abort + ACK



Coordinator



Participant

Green arrows are forced-writes. Black arrows are non-forced writes.

> 2PC is synchronous within 1 state transition, i.e. no site leads another site by more than 1 state transition during 2PC execution.

Recovery Protocol

fails in	Coordinator	Participants
INITIAL	Resumes in INITIAL state	Aborts xact unilaterally
WAIT/ READY	Send PREPARE messages again	Sends Vote-Commit to Coordinator
COMMIT/ ABORT	Sends Global decision to all	Do nothing

Basic Termination Protocol

timeout in	Coordinator
WAIT	Waiting for votes; Writes an abort record in log; Sends Global-Abort
COMMIT/ ABORT	Waiting for ACK; Sends Global decision to unresponsive participants

timeout in	Participants
INITIAL	Waiting for PREPARE; Abort xact unilaterally
READY	Waiting for Global decision; Blocked!

Cooperative Termination Protocol

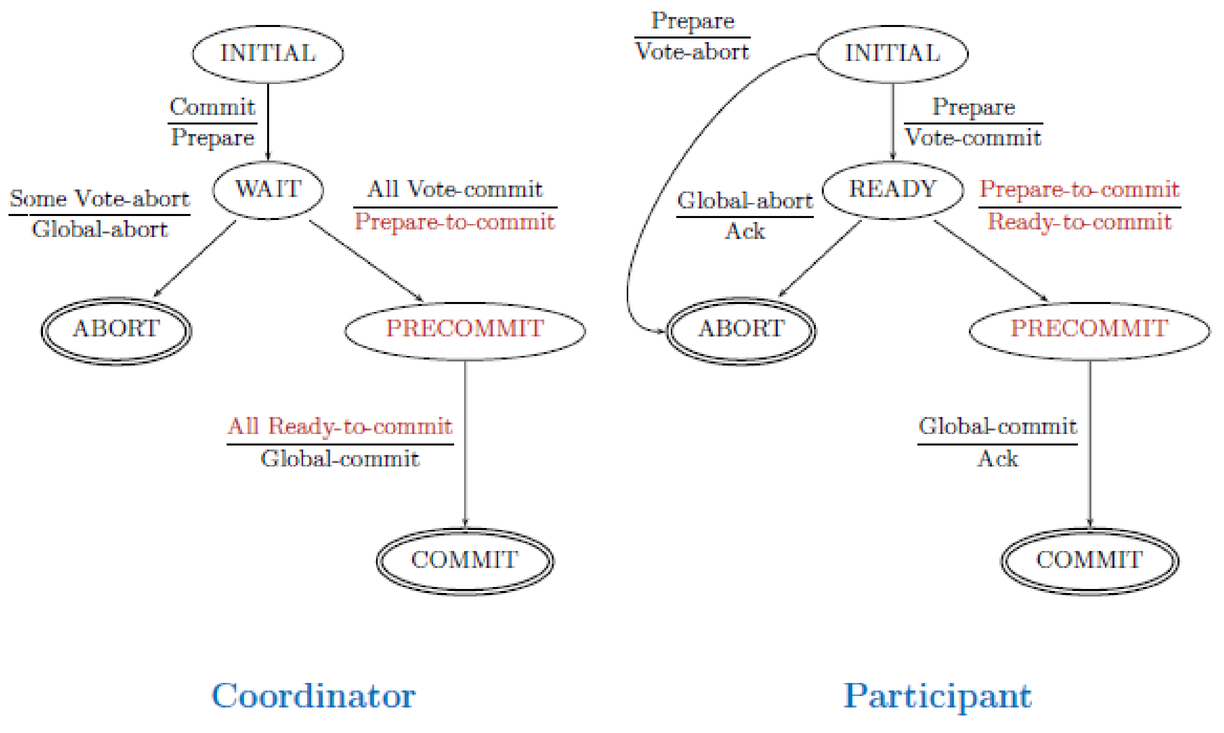
Goal: reduce probability of blocking by failed coordinator by communicating with other participants.

1. Coordinator includes addresses of all participants in PREPARE msg
2. When participant P timeouts in READY state, send DECISION-REQUEST msg to all other participants
3. Upon receiving a DECISION-REQUEST msg, send ABORT if in INITIAL, UNCERTAIN if in READY, and Global decision if in COMMIT/ABORT
4. Terminates the xact with COMMIT/ABORT if not all send UNCERTAIN
5. Inform every participant that replied UNCERTAIN with the global decision.

Three-Phase Commit (3PC)

Guarantees: In the absence of communication failure & total site failure, 3PC is non-blocking

1. Voting phase: PREPARE + Vote-Commit/Vote-Abort
2. Dissemination phase (if there is no ABORT vote): Prepare-to-Commit + Ready-to-Commit
3. Decision phase: Global-Commit/Global-Abort + ACK



3PC uses more forced-write than 2PC

> 3PC is synchronous within 1 state transition **if there is no failure**

Recovery Protocol

fails in	Coordinator	Participants
INITIAL	Asks some participant for xact outcome	Aborts xact unilaterally

fails in	Coordinator	Participants
WAIT/ READY/ PRE-COMMIT	Asks some participant for xact outcome	Asks some participant/ coordinator for xact outcome
COMMIT/ ABORT	Do nothing	Do nothing

Termination Protocol

timeout in	Coordinator
WAIT	Waiting for votes; Writes an abort record in log; Sends Global-Abort
PRECOMMIT	Sends Prepare-To-Commit to unresponsive; Writes commit record to log; Sends Global-Commit to operational
COMMIT/ ABORT	Waiting for ACK; Sends Global decision to unresponsive participants

timeout in	Participants
INITIAL	Waiting for PREPARE; Abort xact unilaterally
READY	Waiting for Global decision; Execute New Coordinator Termination Protocol
PRECOMMIT	Waiting for Global-Commit; Execute New Coordinator Termination Protocol

New Coordinator Termination Protocol

1. Operational participants elect a new coordinator C'
2. C' sends STATE-REQUEST msg to operational participants
3. Terminates the xact:
    1. some in COMMIT → sends Global-Commit to all operational
    2. none in PRECOMMIT → sends Global-Abort to all operational
    3. C' resumes in WAIT state by resending Prepare-to-Commit msgs

Some results:

- Any participant/coordinator X that fails and then recovers while the termination protocol is in progress, will not be allowed to re-participate in the termination protocol.
- It is possible for the global decision to abort even though a participant is in PRECOMMIT state

5 Concurrency Control

Review:

- serial schedule, view equivalent (same results for reads & same final writes), view serializable schedule, conflict equivalent, conflict serializable schedule, recoverable schedule (commit after all deps committed)

- 2PL (grow + release phase) schedules are conflict serializable
- strict 2PL (2PL but hold locks until xact terminates) are conflict serializable and recoverable
- Deadlock: Detection (waits-for graph) + Prevention (wait-die, wound-wait)

MVCC (Multiversion Concurrency Control)

Definition.

- Let $W_i(x)$ creates a new version of x denoted by x_i.
- In a multiversion schedule, if there are multiple versions of an object X, a read action on X could return any version.
- Schedules S and S' are **multiversion view equivalent** if they have the same set of read-from relationships.
- A **monoversion schedule** is a multiversion schedule S that returns the most recently created object version
- A multiversion schedule S is a **multiversion view serializable schedule** (MVSS) if there exists a serial monoversion schedule that is multiversion view equivalent to S.

Theorem. A view serializable schedule (VSS) is an MVSS.

Note: the VSS must be a monoversion.

Snapshot Isolation

Each xact T is associated with a start timestamp and a commit timestamp.

Two xacts are concurrent if their [startTS, commitTS] interval overlaps.

Concurrent Update Property. If multiple concurrent xacts updated the same object, only one of xacts is allowed to commit.

1. First Committer Wins (FCW): check if there exists a committed concurrent xact T' that has updated some object that T modified.
2. First Updater Wins (FUW): updater hold X-lock for all objects O updated. Also needs to check if O has been modified by a *committed* concurrent xact.

Anomalies

SI does not guarantee serializability. Some anomalies:

1. Write Skew Anomaly
2. Read-Only Xact Anomaly

Garbage collection

A version O_i of object O may be deleted if there exists a newer version O_j, i.e. `commitTS(T_i) < commitTS(T_j)` such that every active xact starts after `commitTS(T_j)`

Distributed Transactions

Define

- Originating site: site where Xact is initiated
- Xact coordinator (TC): transaction manager (TM) at originating site
- Local schedule: xact schedule at a local site
- Global schedule: schedule S for which each local schedule S_i is a subsequence of S.

Theorem. A global schedule S is serializable if each local schedule S_i is serializable, and the local serialization orders are compatible.

Lock Based Protocol

Centralized 2PL: one site designated as central site. Xact coordinator makes lock requests / releases to central TM

Distributed 2PL: Locks are managed collectively by each site's lock manager.

Distributed Deadlock Detection

Centralized Approach:

1. Each site maintains a local Wait-For Graph (LWFG)
2. Periodically, each site transmits its LWFG to the deadlock detector. The deadlock detector constructs a global Wait-For Graph (GWFG) and looks for cycles in it.

Snapshot Isolation in Distributed Settings

Key Challenge: How to sync timestamps in distributed environment

Centralized Snapshot Isolation (CSI) → similar to First Updater Wins (FUW)

- One site is designated as Centralized Coordinator (CC), responsible for assigning timestamps (start & commit)
- Write locks are managed collectively (distributed lock)
- To start a new xact T, send a request to CC for : (1) commit timestamp of last commited Xact → used to determine object versions to be read, (2) start timestamp for T
- Modified 2PC Protocol: when commit participants receives a PREPARE msg (which includes `startTS` and `commitTS`), also check for any WW-conflicts between T and all committed concurrent xacts. If there is any, vote ABORT.

6 Data Replication

Key Question: How to keep replicas in sync?

Definition

- An execution is **one-copy serializable** (1SR) if there exists a serial execution on a non-replicated (one-copy) database with the same effect.
- A replicated database is **mutually consistent** if all the replicas of its data items have identical values
  - Strong MC : all copies of data are consistent at the end of update xact

- Weak MC: do not require all copies of data to be consistent at the end of update xact

Replication Methods

DBMS-level replication

- Statement-based replication: foward SQL statements to replica
- WAL shipping: send T's log records to replica sites for sync
  - physical replication: storage-based specs of updates (e.g. location of modified bytes on disk block)
  - logical replication: row-based specs of updates

Application-level replication

Replication Protocols

Assume: strict 2PL Concurrency Control, statement-based replication

When are updates propagated to copies?

- Eager : propagates updates to **all** replicas within context of xact.
  - Enforces strong mutual consistency
  - Always produce 1SR schedules
- Lazy: Xact updates only 1 replica; others are updated asynchronously using **refresh xact**.
  - Weak Mutual Consistency
  - Might produce non-1SR schedules

Where are updates allowed to occur?

- Centralized: Update is applied to a master copy first
- Distributed: Update is applied to any copy

Refresh Xacts

- Refresh xacts T_i^r are to be applied to all slave sites in the same order, by ordering its timestamp $TS(T_i^r)$ = `commitTS(T_i)`

Note:

> Reads always try to read from its own replica if it exists

Eager Centralized Protocols

Eager Single-Master: a protocol where master copies for all obejcts are stored in a single master site. Described below is for the Eager Primary Copy protocol

Key Idea: for each object O, a single master site containing the master copy and its TM functions as the centralized lock manager.

Eager Distributed Protocols

Difference from Eager Primary Copy: query local site's lock manager is enough. Each site runs a lock manager controlling locks for its local replicas.

Lazy Centralized Protocols

Difference from Eager Single-Master:

1. No need to directly execute writes in slaves
2. When T commits, notify master site, who will commit, release locks for T, grants X-locks for T's refresh xacts, and send refresh xacts to slave.

Lazy Distributed Protocols

Combine differences of Lazy Centralized and Eager Distributed :)

Might result in inconsistent updates: multiple xacts update different copies of same data concurrently at different sites

Reconciliation of Inconsistent Updates

Assume updated-values replication, we can use

- Last Writer Wins Heuristic: Ignore T_i^r if we have received a bigger timestamp for another refresh xact updating the same data.
  - works only for blind writes (new value of x is independent of its previous value). Non-blind writes suffer from lost updates anomaly

Quorum Consensus

Previously we assume read from one, write to all (ROWA), but we can vary adjust the read thresholds and write thresholds.

Assign a weight $W(O_i)$ to each copy O_i of object O. Let $W(O)$ be the total weight of all copies of O. Let $T_r(O), T_w(O)$ be the read and write thresholds respectively. The constraints given are:

1. $T_r(O) + T_w(O) > W(O)$
2. $2T_w(O) > W(O)$

The intuitions for the 2 constraints are such that for any conflicting operations, there is at least 1 replica in which the conflicting ops overlaps, and so we can use the order of execution in that replica to deduce the order of execution of the global transactions.

- To read an object O, return the copy with the highest version number among the read quorum
- To write an object O, write all copies to the quorum with version number set to the quorum's highest version number + 1.

Failure handling for single-master replication

Failure of slave sites

- Lazy: sync unavailable replicas later once they're available
- Eager: relax write all constraint to write to all available → and sync unavailable replicas later

Failure of master sites: elect a new master site

1. Choose a partition P to remain available
   1. Simple algorithm: choose P to be the partition that contains an operational master site
   2. Majority Consensus algorithm: choose P where $|P| > \lfloor N/2 \rfloor$
   3. Quorum Consensus algorithm: Assign each replica a non-negative weight. Choose P if the total weight of the partition exceeds half the total weight of all the replicas.
2. If P does not contain an operational master site, elect a new master site in P using some consensus algorithm (see Raft)

7 Raft

Fault-tolerant Replicated State Machine.

Server states, Terms

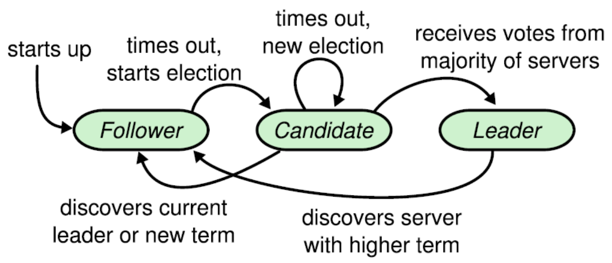

starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

Figure 4: Server states. Followers only respond to requests from other servers. If a follower receives no communication, it becomes a candidate and initiates an election. A candidate that receives votes from a majority of the full cluster becomes the new leader. Leaders typically operate until they fail.

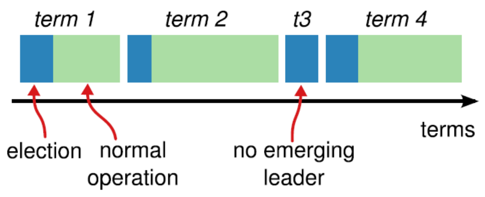


Figure 5: Time is divided into terms, and each term begins with an election. After a successful election, a single leader manages the cluster until the end of the term. Some elections fail, in which case the term ends without choosing a leader. The transitions between terms may be observed at different times on different servers.

Logs, Log Inconsistencies, Log Completeness

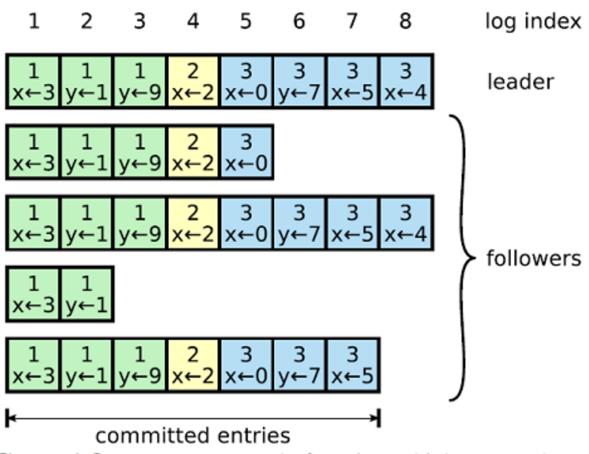


Figure 6: Logs are composed of entries, which are numbered sequentially. Each entry contains the term in which it was created (the number in each box) and a command for the state machine. An entry is considered *committed* if it is safe for that entry to be applied to state machines.

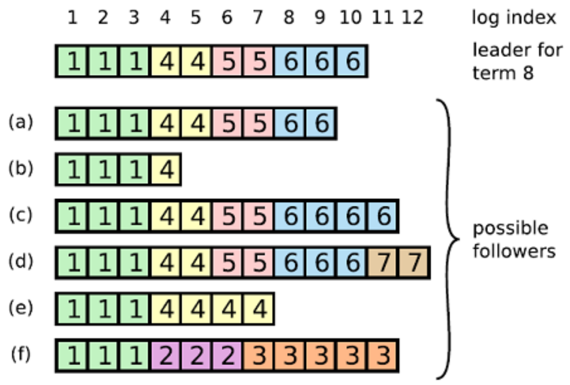


Figure 7: When the leader at the top comes to power, it is possible that any of scenarios (a–f) could occur in follower logs. Each box represents one log entry; the number in the box is its term. A follower may be missing entries (a–b), may have extra uncommitted entries (c–d), or both (e–f). For example, scenario (f) could occur if that server was the leader for term 2, added several entries to its log, then crashed before committing any of them; it restarted quickly, became leader for term 3, and added a few more entries to its log; before any of the entries in either term 2 or term 3 were committed, the server crashed again and remained down for several terms.

X's log is **more complete** than Y's log if `(X.lastLogTerm, X.lastLogIndex) > (Y.lastLogTerm, Y.lastLogIndex)`. In figure 7, d > c > a > e > b > f

Committed Entries

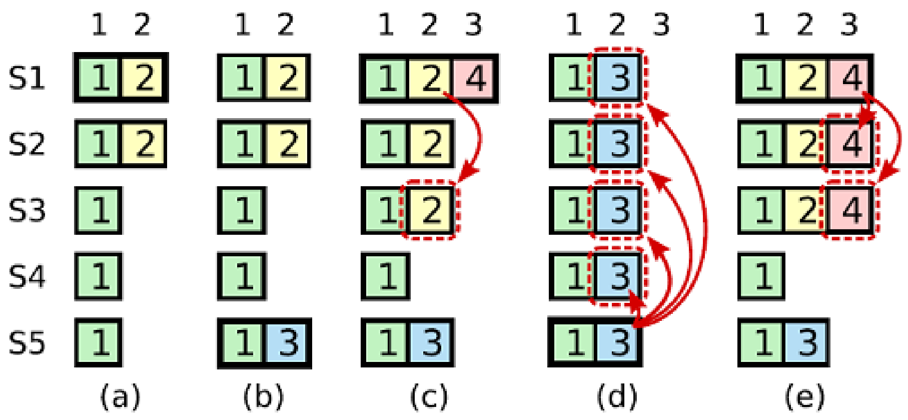


Figure 8: A time sequence showing why a leader cannot determine commitment using log entries from older terms. In (a) S1 is leader and partially replicates the log entry at index 2. In (b) S1 crashes; S5 is elected leader for term 3 with votes from S3, S4, and itself, and accepts a different entry at log index 2. In (c) S5 crashes; S1 restarts, is elected leader, and continues replication. At this point, the log entry from term 2 has been replicated on a majority of the servers, but it is not committed. If S1 crashes as in (d), S5 could be elected leader (with votes from S2, S3, and S4) and overwrite the entry with its own entry from term 3. However, if S1 replicates an entry from its current term on a majority of the servers before crashing, as in (e), then this entry is committed (S5 cannot win an election). At this point all preceding entries in the log are committed as well.

- A log entry is **directly committed** once the leader that created the entry has replicated it to a majority of servers
- All log entries preceding a directly committed entry are indirectly committed

Once an entry is committed:

- Leader execute command in its state machine & return results to client
- Notifies followers of committed entries in subsequent `AppendEntries`
- Followers execute committed commands in their state machines

Leader election

`RequestVote(candidateId, candidateTerm, lastLogIdx, lastLogTerm) -> (term, voteGranted?)`

- Become candidate if it receives no heartbeat over a time period
- Begin election: increment current term + send `RequestVote` to all other servers + chooses an election timeout duration randomly from [T, 2T] (T >> avg send-receive time)
- Wins the election if it receives votes from a majority of the servers
  - Vote for at most 1 candidate (first-come first-serve) in a term→ guarantees Election Safety Property

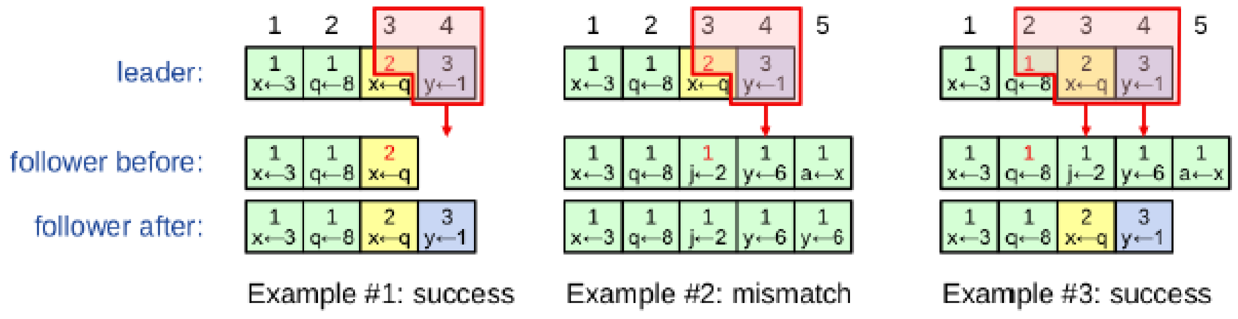
- Votes for candidate C if C's log is *more complete* than its own log AND it has not voted before in the current term → guarantees Leader Completeness Property
- Once elected, establishes authority by sending heartbeat msgs
- If it receives `AppendEntries` from another server with term $\geq$ its own term, recognizes the leader as legit and return to follower state
- If election timer times out, begin new election

Server states

- persistent state: `currentTerm, votedFor, logs[]`
- volatile state on all servers:
  - `commitIndex`: idx of highest log entry known to be committed
  - `lastApplied`: idx of highest log entry applied to state machine
- volatile state on leaders: `nextIndex[], matchIndex[]`
  - `nextIndex[]` stores index of next log entry to send to that server, initialized to leader's last index + 1
  - `matchIndex[]` stores index of highest log entry known to be replicated on server, initialized to 0

AppendEntries

`AppendEntries(leaderId, leaderTerm, leaderCommitIdx, prevLogIndex, prevLogTerm, entries[]) -> (followerTerm, success?)`



- Follower F must contain matching entry; otherwise F will reject `AppendEntries` and leader retries with lower `prevLogIdx`
- Follower F will also update its `commitIndex` to `min(leaderCommit, index of last entry)`

Timers: Election Timer, Leader Timer, Client Timer

Additional Rules

For all servers:

- If `commitIndex > lastApplied` → increment `lastApplied` and apply `log[lastApplied]` to state machine
- If an RPC's term > `currentTerm` → `currentTerm = T`, convert to follower

For leaders:

- Send empty `AppendEntries` upon elected and during idle period
- If received command from client → append entry to local log and respond to client after entry has been applied to state machine
- If `lastLogIndex >= nextIndex[F]` → send `AppendEntries`
- If exists `N > commitIndex` AND a majority of `matchIndex[i] >= N` AND `log[N].term = currentTerm` → set `commitIndex = N`

Properties

1. Election Safety: at most one leader can be elected at a given term
2. Leader Append-only
3. Log Matching: if 2 logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index
4. Leader Completeness: If a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms
5. State Machine Safety: If a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index.

8 Replicated Data Consistency

Levels of consistency

1. Strong consistency: see all previous writes.
2. Consistent Prefix: see an initial sequence of writes
3. Bounded Staleness: (a.k.a bounded CP) see all "old enough" writes
4. Monotonic Reads: see increasing subset of writes (over multiple reads)
   - states read might not exist historically
5. Read My Writes: See all writes performed by reader.
   - non-writer behavious is similar to eventual consistency.
6. Eventual consistency: See subset of previous writes.

Pileus - A deepdive

- KV store, range-partitioned on keys
- replicated using lazy centralized replication protocol (all writes to primary sites and propagated in-order to secondary sites)
- **Distributed Snapshot Isolation** protocol for CC, each xact get a `readTS` and a commitTS (complements discussion in Lec 5)
- In Pileus, each **session** contains 1 or more **transactions**. Scopes for RMW and MR are per session. Each transaction can specify a different consistency level.

Tunable consistency levels

- strong consistency: see all xacts before `readTS`
- consistent prefix consistency: see results of an arbitrary prefix of committed xacts
- read my writes: see the results of all previous Puts in the current session to objects accessed by T
- monotonic reads: `readTS >=` all previous Gets in the current session.
- bounded(T): see all xacts before `readTS - T`

- causal consistency : `readTS >=` all previous Gets & Puts in the current session.
  - T1 < T2 if
    - T2 is executed after T1 in the same session
    - T2 reads some object written by T1
    - T1 & T2 Puts on the same object, and `commitTS`(T1) < `commitTS`(T2)
    - exists T3 s.t. T1 < T3 < T2.

Storage:

- Servers maintain: Maintain `key-range, highTS` (commitTS of last processed xact), `lowTS` (timestamp of most recent pruning operation)
- Old data versions are periodically pruned by increasing `lowTS`. Last version with `commitTS <= lowTS` is kept.
- Client maintain: `key-range[S], latency[S], highTS[S]`, `commitTS` of previous Puts in the session, `commitTS` of versions returned by previous Gets.

Operations

- `Put(k, v)`: Put ops by T are buffered at client and is only visible to T while T is active.
- `Get(k)`, i.e. `Get(k, readTS(T))`.
  - a server S checks whether `readTS` is within its time range, and return `(v, v.commitTS, S.highTS)` where `v.commitTS < readTS(T)`

Choosing `readTS(T)`:

1. Determine `MARTS(T)` (Minimum acceptable read timestamp) ← depends on the consisteny level requested
2. Define a mapping f from k_i in key-set(T) = $\{k_1, \dots, k_n\}$ to its "closest" server S_i, i.e. server with lowest latency that has `highTS >= MARTS(T)`. On a tie, choose higher `highTS` to reduce staleness.
3. Then, `readTS(T) = min { highTS[S_i] | S_i}`

EndTx: details see slide. TLDR: Similar to 2PC with an additional TS negotiation step (prepare-commit → propose TS → agree on max TS → vote-commit/abort, while validating + updating logical clock with `max(local clock timestamp, commitTS + 1)` → commit)

- Commit Coordinator (CC) : one of the primary servers with data updated by T (commit participants)
- `PutSet` is partitioned based on the primary servers.
- `commitTS` is max `proposedTS` from all commit participants.
- pending queue: transaction in the process of committing, added during propose TS step. (`PutSet_i`, `proposedTS`)
- propagating queue: tx added after global decision. new object versions to be asyncly send to secondary servers
- Such pending transactions might block other requests: `Get(k)`, validation request, replicating updates.

9 Query Processing

Cost model:

1. CPU & IO cost of joining relations X and Y
2. Communication cost of sending relation X from one site to another

Query rewriting

- query decomposition
  1. Normalization using Conjunctive NF (CNF) or Disjunctive NF (DNF)
  2. Simplify using relational algebra equivalence
- localization: rewrites distributed query into localized query

Distributed Join Execution

Assume R and S have been partitioned over all the nodes. Consider $R \bowtie_A S$. Let $\|S\|$ be the size of table S in bytes.

1. Collocated/Local Join: if both R and S have been partitioned on A.
   - Cost: 0
2. Directed Join: If one of the tables (wlog R) has been partitioned on A.
   - We dynamically repartition S on join key A. Cost: $\|S\|$
3. Broadcast Join: Replicate table S to all nodes.
   - Cost: $(n-1) \times \|S\|$
4. Repartitioned Join: Neither tables are partitioned on A.
   - Cost: $\|R\| + \|S\|$

10 Query Optimization

Cost model:

- Cost model for each operator's algorithms
- Estimation assumptions:
  - Uniformity assumption: uniform distribution of attribute values
  - Independence assumption: independent distribution of values in different attributes
  - Inclusion assumption: inclusion dependency between join columns
- Database stats: attribute size, tuple size, number of tuples, relation size, number of distinct values of an attribute, attribute's max / min

Size estimates

Result cardinality = Max number of tuples x product of all selectivity

Assumption 1: Uniformly distributed over all distinct values in a column.

1. col = value → sel = 1 / NKeys(I)
2. col1 = col2 (handy for joins too) → sel = 1 / Max(NKeys(I1), NKeys(I2))
3. col > value → sel = (High(I) - value) / (High(I) - Low(I) + 1)

Assumption 2: Independent Predicates

- Selectivity of AND = product of selectivities of predicates

- Selectivity of OR = sum of selectivities of predicates - product of selectivities of predicates
- Selectivity of NOT = 1 - selectivity of predicates

Assumption 3: Preservation of value sets

For joins $U = R1(A, B) \bowtie R2(A, C)$, then

- NKeys(U, A) = min { NKeys(R1, A), NKeys(R2, A) }
- NKeys(U, B) = min { | U | , NKeys(R1, B) }.

Semijoin Optimization

- A tuple $t \in R$ is a **dangling tuple** w.r.t. $R \bowtie S$ if t does not join with any tuple in S.
- $R \ltimes_A S$ eliminates dangling tuples in R
- $R \ltimes_A S = R \bowtie_A \pi_A(S) = \pi_{\text{attribute(R)}}(R \bowtie S)$
- Selectivity of semijoin $R \ltimes_A S$ = NKeys(S, A) / range domain of A.

Total Cost: CPU Cost + I/O Cost + Communication cost (number of msgs + size of transferred data)

Suppose we plan to do $R \bowtie_A S$. To minimize size of transferred data, we often do a local semijoin first. A semijoin is beneficial if cost of sending $\|\pi_A(S)\| + T_{\text{msg}}$ < not sending dangling tuples : $\|R\| \times (1 - SF(R \ltimes_A S))$. (SF is the selectivity factor)

SDD-1 Algorithm

Semijoin-based approach to minimize total communication cost.
Idea:

1. Generate all possible semijoin reductions.
2. Iteratively select a sequence of semijoin reductions starting from the most beneficial (compare benefit-cost) while updating stats (multiplying size, NKeys with sel)
3. Determine assembly site to compute joins (compare communication cost)
4. Post-optimization: eliminate unnecessary semijoin reductions
5. Execute plan.

Full Reducer

Motivation: Given a join query Q on a database D, is there a *fixed* sequence of semijoins to eliminate all dangling tuples wrt Q for *any* instance of D?

Such sequence of semijoins S is called a **full reducer** for Q.

We can represent a query as a **hypergraph**:

- each node represents an attribute
- each hyperedge represents the set of attributes in a relation.

- A hyperedge e is an **ear** if there exists some other hyperedge e' s.t. every node in e is either found only in e, or also found in e'. In this case, we say e' **consumes** e**.**
- An **ear reduction** is the elimitation of 1 ear from the hypergraph along with any nodes that appear only in that ear.
- A hypergraph is **acyclic** if it can be reduced to a single hyperedge by a sequence of ear reductions.

Theorem. A full reducer exists for a join query iff its hypergraph is acyclic.

```
FullReducer(Acyclic_Hypergraph G)
    If | G | = 1: return.
    Let e be ear in G consumed by e'
    e & e' corresponds to relation R & R'
    yield R' SEMIJOIN R
    FullReducer(G' = G with e removed)
    yield R SEMIJOIN R'
```