Partitioning

Completeness	Given a fragment U derived from R, all keys in U must be in R
Disjointness	Given a fragment U derived from R, all keys in U must be unique foreign key in R
Reconstruct	R = $R_1 \cup R_2 ... \cup R_n$
Range partitioning	$R_1 = \sigma_{A<100}(R)$, $R_2 = \sigma_{A\in[100,500]}(R)$, R = $R_1 \cup R_2$
Hash partitioning	A record is stored in node N if h(t, A) falls in N's region, adding a new node only affects on node unlike modulo method, but is oblivous to servers' capacity
Derived horizontal partitioning	$R_i = R \ltimes_A S_i$, each S_i is a partition of S
	for R to be complete all A in R must exist in S
	for R to be disjoint, S.A must be unique key
	for R to be disjoint and complete, R.a must be a foreign key of S with non-null values of R.a
Complete partitioning	F is complete partitioning of R wrt Q, if for all fragment $R_i \in F$, either every tuple in R_i matches Q or every tuple does not match Q
MTPred	$m_i = p_1^* \wedge p_2^*, MTPred(P) = m_0, m_1, ...m_7$
	e.g. $m_0 = \neg p_1 \wedge \neg p_2 \wedge \neg p_3$
	$S_i = \sigma_{m_i}(R)$, we say S_i is the min term predicate partitioning of R wrt Q and F = $\{S_1, ..., S_m\}$, F is a complete partitiong wrt every query in Q

Query processing

RA equivalence

Commutativity of binary ops	$R \times S = S \times R, R \bowtie S = S \bowtie R$
Associativity of binary ops	$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$
Idempotency of unary ops	$\pi_{L'}(\pi_L(R)) = \pi_{L'}(R)$, anti-intuitive reprojection to push across join
	$\sigma_{p_1}(\sigma_{p_2}(R)) = \sigma_{p_1 \wedge p_2}(R)$, separate selection
Commutating sel and proj	$\pi_L(\sigma_p(R)) = \pi_L(\sigma_p(\pi_{L \cup attr(p)}(R)))$
Push down selection	
Push down projection	

Join strategies

Strategy	Comm cost	When
Collocated	0	All partitions on the same key in all tables
Directed	size(R) if R is being repartitioned	only 1 table is not partitioned on the key
Repartitioned	size(R)+size(S)	both are not partitioned on the key
Broadcast	$(n-1) \times size(R)$ if R is broadcasted, R is smaller in size	both are not partitioned on the key

NOTE: If need to compute join from J_1 to other relation, need to see what strategy used previously. For eg. S is partitioned on B, R on C and T on F. Use broadcast join by broadcasting R for $J_1 = R \bowtie_A S$. Now, $J_1 \bowtie_B T$ only requires directed join on T since J_1 is partitioned on B already.

Storage

LSM storage

- improves write thoughput by converting random I/O to sequential I/O
- append only updates
- vatch insert to disc once mem is full
- slow down read query coz need to search both mem and disk

MemTable

- updated in-pace
- flush records to disk when reaches a threshold
- SSTables are immutable and are assoc. with a range of ket values and a ts
- each new update is appended to commit log and updated to MemTable

Size-tiered compaction strategy

- SSTables in each tier have approx same size
- compaction happens when no. of SSTables reaches a threshold
- all SSTables in tier L are merged into a single SSTable in L + 1
- L becomes empty after compaction
- S_0 is more recent than S_1, $S_{0,4}$ is more recent than $S_{0,1}$

Level-tiered compaction strategy

- SSTables at L = 0 may not be disjoint
- SSTables at L > 0 have same size, disjoint
- SSTable at L iverlaps with at most F SSTables at L + 1
- compaction at L = 0, merge all SSTables with all overlapping SSTables at L + 1
- compaction at L > 0, round-robin
- NOTE: $F^{L-1} < size(L) \leq F^L$, compacts when size(L) > F^L. Each level stores F times as much data as previous level.
- optimize with sparse index using block's smallest key value then search with B+ tree; Bloom filter with h(x) to quickly check if a seach key exists in a SSTable block

Max size of R is mn MB. $F^{L-1} < mn \leq F^L$, using log we have $L = [log_F(mn)]ceil$. Increase F -> smaller L -> improves worst case I/O cost for searching but more I/O costs when merging SSTables.

Local indexing

- each server keeps a table to store mapping
- redundant index across servers, some sort of scatter-gather query
- easy to update since in the same server, not so good for searching

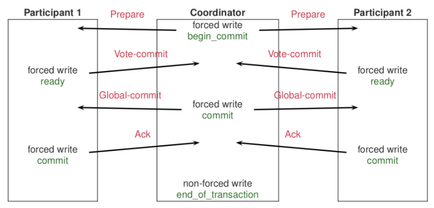
Global indexing

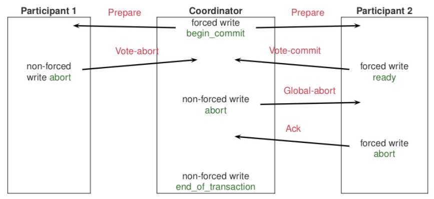
- every server keeps a metadata to record each index is stored at which server, partitioning done by hashing
- to search a key, go to the server with the index, get the mapping, may need to go to different servers to fetch the actual records
- good for search but update is more expensive if the mapped record is stored in different server

NOTE: To check using which index is more efficient, check no. of index partition and relation partition need to be accessed

Commit Protocol

2PC protocol





Recovery of Coordinator

Fails in state	Recovery actions
INITIAL, WAIT	Writes abort log record, sends Global-Abort to all
ABORT	Does nothing if received all ACK, else sends Global-Abort to all
COMMIT	Does nothing if received all ACK, else sends Global-Commit

Recovery for Participants

INITIAL	unilaterally abort
READY	sends Vote-Commit to TC, this is dependent
ABORT, COMMIT	Does nothing

Basic Termination for Coordinator

timeout in state	terminate actions
WAIT	Writes abort log record, sends Global-Abort to all
ABORT	sends Global-Abort to non-responding participants
COMMIT	sends Global-Commit to non-responding participants

Basic Termination for Participants

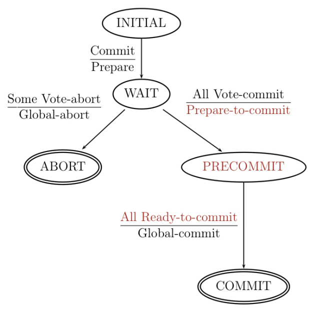
timeout in state	terminate actions
INITIAL	aborts unilaterally
READY	Blocked! Need to wait TC to recover first

Cooperative termination protocol - Participant Q receives a Decision-Request from Participant P that timeouts.

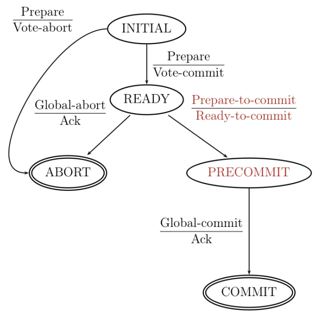
Receive in state	Replies actions
INITIAL	aborts unilaterally, "Aborts"
READY	"Uncertain"
COMMIT	"Commit"
ABORT	"Abort"

Can still be blocking if the remaining participants are all in READY state.

3PC Commit Protocol



Coordinator



Participant

3PC recovery protocol for TC and P

Recover in state	Recovery actions
INITIAL	aborts unilaterally
WAIT or READY, PRECOMMIT	asks other TM
COMMIT, ABORT	Does nothing

Termination protocol for coordinator

WAIT	Writes abort log, sends "Global-abort"
PRECOMMIT	Writes commit log, sends "Global-commit"
COMMIT, ABORT	Sends "Global-commit" or "Global-abort" to those not yet responded

Termincation protocol for P

INITIAL	Unilateral abort
READY,PRECOMMIT	Executes 3PC1

3PC termination protocol 1 - Participants execute this when they timeout in READY or PRECOMMIT state

- TMs elect new TC
- TC sends State-Request msg to all TMs
- Each TMs responds to TC about its current state
- If some TMs in COMMIT, TC sends "Global-Commit" to all
- else if no TMs in PRECOMMIT, TC sends "Global-Abort" to all
- else (means some in READY some in PRECOMMIT), TC sends "Prepare-to-commit" to READY TMs. Sends "Global-commit" after receiving "Ready-to-commit" from these TMs.

NOTE: Failed TMs are not allowed to join Termination protocol to prevent inconsistent global decision. 3PC1 is correct if there is no connection failure. 3PC1 is blocking but correct in the event of total site failure, need to wait for a TM to recover such that it can recover independently(in INITIAL, ABORT, COMMIT) or it was the last TM to fail. The former would notify using its recovered state, the latter would execute termination protocol. In the event of comm failure, execute 3PC2.

3-PC termination protocol 2 - to ensure correctness when comm failute

- TMs elect new TC
- TC sends State-Request msg to all TMs
- Each TMs responds to TC about its current state
- If some TMs in COMMIT, TC sends "Global-Commit" to all
- else if some in ABORT, TC sends "Global-Abort" to all
- else if majority in READY/PRECOMMIT, same as 3PC1. If PRECOMMIT and Ready-to-commit forms majority, sends "Global-commit". Otherwise TC and TMs are blocked.
- else if majority in READY/PRECOMMIT/PREABORT, sends "Prepare-to-abort. If PREABORT and "Prepare-to-abort" forms majority, sends "Global-abort". Otherwise, TC and TMs are blocked.
- else TC and TMs are blcoked

NOTE: 3PC2 is correct when total site failure and comm failure, is non-blocking in the absence of comm failure as long as majortiy of TMs are still operational

Concurrency Control

View equivalent	Same read from and final Write
Conflict equivalent	Same ordering of conflicting pairs
View/Conflict serializable	S is view/conflict equivalent to a serial schedule
Serializable global schedule	each local schedule is view/conflict serializable -> exists a local serial schedule
	Each local serial schedule is a subsequence of an overall global serial schedule -> each local serial schedule is compatible ordering wise
monoversion schedule	each read action returns the most recently created object version
multiversion view equivalent	Read from only, ignore final write. $W_i(x)$ creates a new version of x denoted by x_i, x_0 is initial version
serial multiversion schedule	a multiversion schedule that is multiversion view equivalent to a serial monoversion schedule
recoverable schedule	if T reads from T', T must commit after T' for S to be recoverable

NOTE MVSS is not necessarily VSS, but VSS must be MVSS

2PL Lock based CC protocol

- To read O, a txn must holde a S-lock or X-lock on O
- To write to O, a txn must hold X-lock on O
- A txn is not allowed to request for any more locks once it releases a lock, thus 2PL is guaranteed conflict serializable

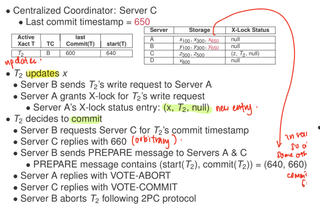
- Strict 2PL says a txn must hold on to locks until it commits or aborts, thus S2PL is guaranteed recoverable
- Deadlock is detected using wait-for graph or timeout mechanism.
- Wait-die policy: non-preemptive, txn gets aborted only when reqesting for a lock, a txn with all the locks it needs is never aborted and younger txns may get aborted repeatedly
- wound-wait: preemptive
- lock-conversion to allow for interleave schedule, but need to check before granting conversion

Multiversion based CC protocol
- Benefits: Read-only txns never get aborted and are not blocked by update txns, update txns are never blocked by read-only txns
- Concurrent txns: $[start(T), commit(T)] \cap [start(T^{'}), commit(T^{'})] \neq 0$
- Only allow one txn to commit if multiple concurrent txns updated the same object to ensure serializability. (Disjoint write-sets)
- First-commit win vs First-update win. In FUW if x-lock is not held by $T^{'}$, txn is granted X-lock. Txn aborts if O has been updated by concurrent txns, otherwise txn proceeds. If X-lock is held by $T^{'}$ and it has aborted, txn is granted X-lock and same as previous. If $T^{'}$ commits, txn is aborted. Txn releases X-lock after commit or abort.
- SI does not guarantee MVSS, must use serializable SI for that

Centralized vs Distributed 2PL

	C2PL	D2PL
Lock management	CC even if it doesn't store the object	Managed by each site's lock manager
Comm cost	ReqLock and RelLock to CC, read req to site storing object, read res to coord TM	Read req to site A storing O, site A grants S-lock and Read res to coord TM, coord TM sends RelLock to site A
Deadlock	Aware of all deadlocks	central deadlock detector to construct global WFG from local WFG sent by each site

Centralized Snapshot Isolation
- One site as CC, say site A stores x
- Start a new txn T: TC req start(T) and lastCommit(T)
- $R_i(x)$: TC sends lastCommit(T) to TM_A, TM_A responds with latest version wrt lastCommit(T)
- $W_i(s)$: TC sends write req to TM_A, TM_A checks if X-lock can be granted and responds accordingly, T is blocked if cannot get lock
- Commit(T): TC sends req to CC for commit(T). TC execs 2PL with start(T) and commit(T). When P receives "PREPARE", it checks for WW-conflicts btwn T and committed concurrent Txns, responds "Vote-to-abort" if found concurrent version

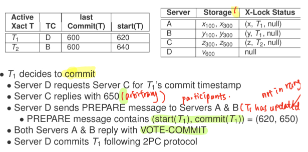


- Centralized Coordinator: Server C
  - Last commit timestamp = 650

Active Xact T	TC	last Commit(T)	start(T)
T_2	B	600	640

Server	Storage	X-Lock Status
A	$x_{100}, x_{300}, x_{650}$	null
B	$y_{100}, y_{300}, y_{650}$	null
C	z_{300}, z_{500}	(z, T_2, null)
D	v_{600}	null

updates .
- T_2 updates x
  - Server B sends T_2's write request to Server A
  - Server A grants X-lock for T_2's write request
    - Server A's X-lock status entry: (x, T_2, null) new entry .
- T_2 decides to commit
  - Server B requests Server C for T_2's commit timestamp
  - Server C replies with 660 (arbitrary) .
  - Server B sends PREPARE message to Servers A & C
    - PREPARE message contains (start(T_2), commit(T_2)) = (640, 660)
  - Server A replies with VOTE-ABORT
  - Server C replies with VOTE-COMMIT
  - Server B aborts T_2 following 2PC protocol



Active Xact T	TC	last Commit(T)	start(T)
T_1	D	600	620
T_2	B	600	640

Server	Storage	X-Lock Status
A	x_{100}, x_{300}	(x, T_1, null)
B	y_{100}, y_{300}	(y, T_1, null)
C	z_{300}, z_{500}	(z, T_2, null)
D	v_{600}	null

- T_1 decides to commit
  - Server D requests Server C for T_1's commit timestamp
  - Server C replies with 650 (arbitrary)
  - Server D sends PREPARE message to Servers A & B (T_1 has updated participants.
    - PREPARE message contains (start(T_1), commit(T_1)) = (620, 650)
  - Both Servers A & B reply with VOTE-COMMIT
  - Server D commits T_1 following 2PC protocol

Replication

Mutual consistency and 1-copy serializability

S_{RD} is equivalent to S_{1C} if
- T_j reads x from T_i in S_{RD} iff T_j also reads x from T_i in S_{1C} and
- for each final write $W_i(x)$ in S_{1C}, $W_i(x)$ is a final write in S_{RD} for some copy x_A of x. MC if all copies of x.

Eager	**Lazy**
Strong MC	Txn commits as soon as one replica is updated
ROWA	Updates other replicas asynchronously using refresh txn
1-copy serializability is guaranteed	Updates from different txns can conflict
	Need to ensure updates are applied in same order

For all protocols, we assume strict 2PL and statement propagation

Centralized	**Distributed**
each primary or master copy site runs a lock manager for S2PL	each site runs a lock manager, lock is granted to local replica, not all copies of an object
Update is applied to master copy first before propagating to slave copies	Update is applied at any site then propagated to other replicas
If need latest data, must read master copy	may get stale value
Master/Primary site is bottleneck	Deadlocks are more common since lock is not managed by a single master copy
Write must be done on master site first, read can be done locally	Prioritize local copy for write
S2PC runs among involved primary sites	S2PC runs among sites that were updated

To determine if schedules could be produced by which protocol, do the following steps:
1. Check for refresh txn, if yes then lazy
2. Check for 1-copy serializability, if no then cannot be eager
3. Check where txn is issued. Then check where reads and writes are performed for centralized or distributed

Different operations using different protocols: 1. $R_i(x)$:
Eager Centralized and Lazy Centralized (Assuming T_i is issued at S_B, primary copy at S_A)
- TM_B requests for S-lock from TM_A. Blocked if not granted
- Read locally if local copy exists, else sends read to other site

Eager Distributed and Lazy Distributed
- If local copy exists, request S-lock locally. Blocked if S-lock not granted. Read locally
- Send read to other site, say S_A. If S-lock granted, value returned from S_A

2. $W_i(x)$:
Eager Centralized and Lazy Cnetralized
- TM_B requests X-lock from TM_A. Blocked if not granted
- TM_B executes $W_i(x)$ and notifies TM_B lock is granted
- If Eager, TM_B executes $W_i(x)$ and sends $W_i(x)$ to TM_C. Ignore this step for lazy protocol

Eager Distributed
- if has local copy, requests for X-lock locally. Blocked if not granted.
- TM_B executes $W_i(x)$ locally and sends $W_i(x)$ to TM_A and TM_C.
- TM_A and TM_C grants X-lock and executes $W_i(x)$. Blocked if not granted.
- if no local copy, sends $W_i(x)$ to site with x, say S_A.
- TM_A grants X-lock to execute $W_i(x)$. TM_A notifies TM_B and sends $W_i(x)$ to TM_C

Lazy Distributed
- if has local copy, requests for X-lock locally and executes $W_i(x)$ locally. Blocked if not granted.
- if no local copy, sends $W_i(x)$ to site with x, say S_A.
- TM_A grants X-lock to execute $W_i(x)$.

3. $Commit_i(T)$:
Eager centralized and Eager distributed run 2PC among the servers involved.
Lazy Centralized:
- TM_B sends commit(T_i) to TM_A.
- TM_A executes commit(T_i) and releases locks for T_i.
- TM_A checks if X-locks can be granted for T_i's refresh txns
- if granted, TM_A sends refresh txns to other sites to propagate T_i's updates. Otherwise propagation of refresh txns is blocked
- Slave sites must ensure refresh txns are applied in the same order using commitTS. This is trivial since TM_A should know the commitTS

Lazy Distributed:
- if TM_B executed $W_i(x)$ locally, TM_B will execute commit
- TM_B releases X-lock for T_i, and sends T_i^r:$W_i(x)$ to TM_A and TM_C
- TM_A and TM_C each grants X-lock for $W_i^r(x)$ and executes $W_i^r(x)$.
- Important to have reconciliation of inconsistent updates.
- Use Las-Write heuristics (aka timestamp order heuristic) to apply updates. Only works for blind-writes
- If 2 concurrent updates $W_i(x)$ and $W_j(x)$, $W_i(x)$ wins if TS(T_j^r) < TS(T_i^r)
- $W_j(x)$ will be ignored by TM. This will not work if $W_j(x)$ is not a blind write.

Handling failure
- Detect site failures with timeout mechanism
- If slave sites fail:
  - Lazy replication: Sync unavailable replicas later when they are available
  - Eager replication: ROWA cannot terminate if one replica is down. Use ROWAA (ROWA Available) to only update available replicas and terminate txn -> sync unavailable replicas when they are available
- If master site fail:
  - Wait for master to recover -> bad availability
  - Elect new master -> might have split brain problems in the event of network partition
  - CAP theorem: When there is network partition, forfeit consistency or availability, can't have both.

Quorum consensus protocl
- $T_r(O) + T_w(O) > Wt(O)$
- $2 \times T_w(O) > Wt(O)$
- To read an object O. acquire S-lock on a read quorum $Q_r(O)$ where $Q_r(O) \geq T_r(O)$, read all copies in Q and return highest version copy
- To write an object O, acquire X-lock on a write quorum $Q_w(O)$ where $Q_w(O) \geq T_w(O)$, write all copies in Q and update their version to n + 1
- For a system with n servers to be k-tolerant where k server has failed, w is the weight of each server:
  - $(n-k)w \geq max(T_w(O), T_r(O))$ for the system to be functional
  - Solve for n to know the min number of servers for k-tolerant

Consistency level
In Pileus, we use 2PC commit protocol and distributed SI protocol concurrency control To determine readTS of a txn:
- Client selects a MART based on the desired consistency level
  - Monotonic read: Max commitTS of **All previous Get regardless of object** in current sesh
  - Read-my-writes: Max commitTS of previous put of object in current sesh
  - Causal: Max commitTS of put and get regardless of objects in current sesh
  - Eventual: 0
  - Bounded staleness: realTimeToLogicalTime(client's clock time - realTime)
  - Strong: Max CommitTS of latest version of all objects in all sesh
- Get servers that has highTS geq MART(T) of all objects
- Each object, select the server with lowest latency, if tie then select the server with larger highTS
- readTS = min of all highTS

End txn commit protocol
- client selects a commit coordinator among participants, where participants are pServers with data updated by txn
- client sends commit req to CC using readTS(txn), Put-set (PS), Largest commit timestamp (LCT) among all PS
- CC updates its localClock = max(localClock, LCT+1), and sends "Prepare-to-commit" with PS_i to P_i
- P responds with proposedTS and appends (proposedTS, PS) to pending list
- CC picks largest proposedTS as commitTS(txn), and sends commitTS to all P.
- P checks commitTS against SI protocol for concurrent updates, responds "Vote-to-abort" or "Vote-to-commit"
- CC force-write log before sending "Global-commit" and all PS, or non-force-write for "Global-abort. CC informs client that txn has committed.
- P_i processes PS_i by creating new version of object upon receiving "Global-commit" and appends to propagate list. Sends "ACK" to CC, removes txn's entry from pending list, asynchronously propagates refresh txn to secondary servers from propragte list.

Operations blocked by pending txn $T^{'}$ in pServer:
- Get(k) by T if $T^{'}$ has updated k and $T^{'}$'s proposedTS $\leq$ readTS(T) -> commitTS($T^{'}$) leq readTS(T)
- both have updated the same key and $T^{'}$'s proposedTS $\leq$ commitTS(T) -> commitTS($T^{'}$) leq commitTS(T)
- Replicating updates of T for propagation, same reasoning as above.

Raft Consensus algorithm
- 2PC and 2PL use RSM to replicate log files. i.e. when a coordinator crashes in 2PC, we can use RSM to replicate log file to other replicas for fault tolerance. Centralized lock manager maintains a lock table for 2PL, can be replicated for fault tolerance and prevent blocking

Terms
- each term starts with election
- at most one leader can be elected per term
- each server maintains **currentTerm, votedFor, log[(index,term,command)]**
- server replies RPC with its term and updates its current term number upon receving larger term number
- Election safety and liveness: each follower can only vote once per term. Some leader must eventually be elected, thus election timeout >> broadcast time

RPC and Timer
- More complete log: X is more complete than Y
  - X.lastLogTerm > Y.lastLogTerm OR
  - (X.lastLogTerm = Y.lastLogTerm) AND (X.lastLogIdx > Y.lastLogIdx)
- RequestVote(candidateId,term,lastLogIdx,lastLogTerm) (term,voteGranted)
- Election timer, Leader Timer, Client Timer: If latency is expected to increase for i.e. geographical latency, need to increase timer as well
- RequestVote is success iff (P = R's log is not more complete than sender's):
  - RPC.term > R.currentTerm AND P
  - RPC.term = R.currentTerm AND R.votedFor = null AND P
  - R.votedFor = RPC.candidateId AND RPC.term = R.currentTerm
  NOTE: If RPC.term > R.currentTerm but $\neg P$, then R.currentTerm = RPC.term but R.votedFor = null. These rules ensure leader's log is at least as complete as a majority of servers'
- Log matching property: 2 entries with same index and term:
  - store the same command
  - preceding entries are identical
- Leader only executes commited log. Log is deemed committed if the **leader that created it** has replicated the log to majority of servers
- Volatile states of all servers: commitIdx, lastAppliedIdx; All leaders: nextIdx[], matchIdx[]
- AppendEntries(leaderId,leaderTerm,leaderCommit,prevLogIdx,prevLogTerm, entries[]) (term,success)
- F processes AppendEntries if not a heartbeat msg:
  - set F.currentTerm = leaderTerm if leaderTerm > currentTerm
  - if F's does not contain prevLogIdx and prevLogTerm, reply (currentTerm,false) so leader can back off and send older logs over
  - else, delete conflicting entries and append logs
  - set F.commitIdx = min(leaderCommit, F.lastLogIdx) if leaderCommit > commitIdx
- Rules for all servers:
  - commitIfx > lastApplied, increment lastApplied by 1 and apply log[lastApplied] to RSM
  - step down to follower if received term > currentTerm
- Rules for leaders:
  - Only respond to client after commitment and execution of command
  - if AppendEntries is successful, update nextIdx and matchIdx. Otherwise decrement nextIdx and resend older entries
  - set commitIdx = N if logs are replicated to majority of server, such that N > commitIdx, matchIdx[i] $\geq$ N and log[N].term = currentTerm
- Election safety, Election liveness, Leader Append-Only (never deletes its own logs), Log matching, Leader Completeness, State machine safety(once executed, no other servers will apply different log entry at the same index)

Query optimization
$SF(\sigma_p(R)) = \frac{card(\sigma_P(R))}{card(R)}$ $SF(R \bowtie S) = \frac{card(R \bowtie S)}{card(R) \times card(S)}$

$SF(R \ltimes S) = \frac{card(R \ltimes S)}{card(R)}$

Join selectivity estimation $SF(R \ltimes_A S) \approx \frac{1}{max(card(\pi_A(R)), card(\pi_A(S)))}$

$SF(R \ltimes_A S) \approx SF_{SJ}(S.A) = \frac{card(\pi_A(S))}{|domain(A)|}$

Optimizing total cost
$R \ltimes_A S$ removes dangling tuple in R wrt to $R \bowtie S$. Beneficial semijoin happens if benefit > cost where,
$Cost(R \ltimes_A S) = T_{msg} + T_{tr} \times size(\pi_A(S))$ to send $\pi_A(S)$ over to site R
$Benefit(R \ltimes_A S) = T_{tr} \times size(R) \times (1 - SF_{SJ}(S.A))$

Comparable plans
1. both plans have the same output schema
2. final operator for both plans is executed at the same site3.
3. both plans output are either unordered or sorted in the same order

Basic vs Enhanced DP
1. basic does not allow for semijoin
2. enhanced requires more processing to handle semijoin:
- Enumerator extension since left and right operand might not be disjoint
- Avoid redundant join and semijoin (every leaf to root path appears once only)
- Fix-point iteration to make plans complete
- Vertical pruning such as inter and intra-entry pruning, as comparable plans may be stored across different optPlan() entries