Weekly Research Progress Report

Student: Sungho Hong **Date:**5/26/2020 **# of hrs worked this week:** 30

Problems discussed in last week's report

1. Reading materials

a. Background of Concurrency Control
b. Serial Safety Net Algorithm

2. Answering questions

a. Understand the algorithms and what it achieves
b. Know how algorithms achieve the goal?
c. How the algorithms can be scaled into a distributed model

Answers

Answering Questions

- The goal of SSN is to improve the performance of CC algorithms
    - Relaxes the strictness of CC algorithms (2PL, RC, SI) by ignoring harmless conflicts that are generally aborted by the CC algorithms
    - Enforces light-weight policy that computes from only the immediate successors of a transaction

- **Figure C:** The SSN achieves the goal by following the exclusive window policy.
    - Aborts π(T) < η(T)
    - π(T) The lowest commit time of the transaction found from the back-edges
    - η(T) The commit time of the most recently committed transaction from the forward-edge

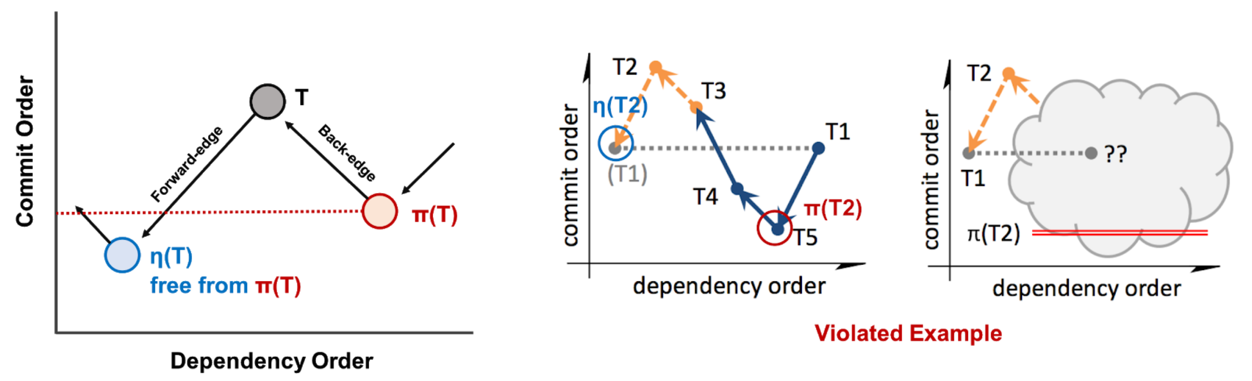


Figure C: Success example (left) & Violated example (right) T1 violates T2's exclusion window because π(T2) = c(T5) < c(T1) = η(T2). The rightmost chart depicts information that checking violation is possible with T2's local knowledge.

- Extending SSN to distributed model
    - The SSN does not assume network communication overhead
        - The SSN assumes the snapshot versions are stored in a memory of a single machine.
        - In distributed systems, additional overhead of tracking down the versions in the remote machine needs to be considered.
    - The SSN is only tested on optimistic CC mechanisms
        - The SSN requires to test the performance of both pessimistic and optimistic methods in a distributed environment.
    - The SSN does not have the propagation policy for replicated data

- For example, client-centric consistency models apply read_set and write_set to maintain consistency of the versions.
    - The client can track the missing versions when the local replica does not have the snapshot version.
    - The client can retry the requ
    - est to another replica that may have the desired snapshot version.

Follow-up Questions

1. Is the project focused on the in-memory database?
    a. Should I focus on the concurrency control of DBMS?
    b. Should I focus on understanding the Latch-based and compare & swap?
2. What is the SI solution that I need to follow?
    a. There are different branches of SI, are there specific versions that you require me to focus on?

Background of CC

- In-Memory Database
  - Relies primarily on memory for data storage
    Pro: Eliminate the need to access disks
  - Con: Cost of transaction of acquiring a lock = Cost of accessing data

- CC (Concurrency Control Algorithm)
  - Allows transactions to access database in multi programmed fashion
  - Preserve the illusion that each of them is executing alone on a dedicated system

- Trade off between strictness and performance of Concurrency Control algorithms
  - Strict policy (2PL, SSI)
    - Forbid many valid serializable schedules.
    - 2PL (Pessimistic) : Lock & change the current database state
    - MVCC, SI (Optimistic): First perform changes in a protected area & then change current database state

- Optimistic MVCC (Multi Version Concurrency Control)
  - The DBMS maintain multiple physical versions of single object in the database
  - Writers don't block readers
  - Readers don't block writers
  - SI (Snapshot Isolation)
    - When a transaction starts, it sees a consistent snapshot of the database that existed when it started.
    - Write-Skew anomaly
  - SI Example: HEKATON MVCC
    - Transaction Lifecycle
      - Simulation Phase
        - Normal Processing
          - track txn read, scan, and write set
      - Validation Stage
        - Validation
          - validate reads and scans
          - If everything okay write new versions
      - Commit Stage
        - Post Processing
          - Update version timestamps
    - Transaction metadata
      - Read set
        - Physical versions that the transaction accessed
      - Write Set
        - Physical versions that the transaction created

                - Scan Set
                    - A set of queries to re-execute and check whether it gets the same result
            - Limitations
                - Read/Scan set validation are expensive if the transactions access a lot of data
                    - This depends on the workload
                - Appending new versions hurts the performance due to increased pointers
                    - This part will degrade even further when the memory is located in a distributed environment.
                - Record-level conflict checks may be too coarse-grained and incur false positives
                    - This part can be improved by using SSN.

- Reference
    - [Explanation of Pessimistic and Optimistic CC, Jens Dittrich]
    - [Multiversion concurrency control, CMU lecture series, 2020]

Serial Safety Net (SSN)

Challenges

- Complexity of CC algorithms
    - bugs can lead to subtle problems
    - difficult to detect and reproduce
        - Changes of CC algorithms are not advisable
- Guarantee performance while following the CC algorithm
    - Read Committed (RC) is still the default isolation level in PostgreSQL for performance reasons
- Heterogeneous workloads
    - Read-mostly transactions.
    - Reads of stale records that are not updated recently do not have to be tracked in the transaction's read set.

Goal of SSN

- Relax the validation stage of CC algorithm
    - CC retains control of scheduling and transactional accesses
    - SSN tracks the resulting dependencies
        - Performs a validation test by examining only direct dependencies of the committing transaction at commit time
        - Determine whether it can commit safely or must abort to avoid a potential dependency cycle

Dependency (Figure:A)

- $T_i \xleftarrow{w:x} T$ **read/write dependency**
  - **Ti ← T** (T depends on Ti)
    - T read or overwrote a version that Ti created
    - Ti committed first and T committed next
    - Ti is a predecessor of T
    - T is a successor of Ti
- $T \xleftarrow{r:w} T_j$ **read anti-dependency**
  - **T ← Tj** (Tj depends on T)
    - T read a version that Tj overwrote
    - Tj committed first and T committed next
    - T is a predecessor of Tj
    - Tj is a successor of T

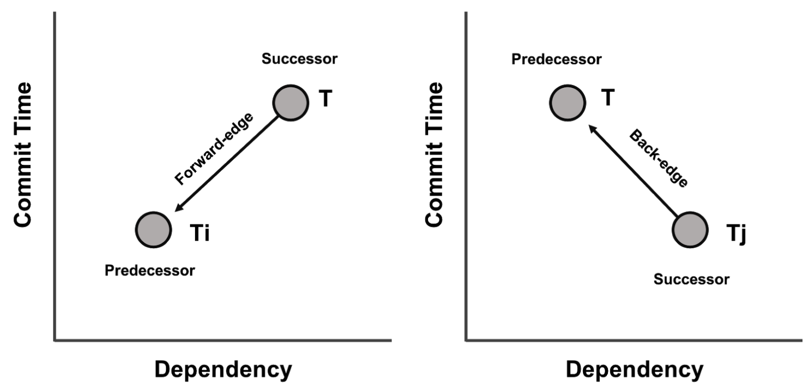


Figure A: read/write dependency (left) & read anti-dependency (right)

Serialization Failures (Figure B)

- $T1 \xleftarrow{w:x} T2 \xleftarrow{w:x} T1$ **Isolation**
    - T1 and T2 saw each other's writes
- $T1 \xleftarrow{w:x} T2 \xleftarrow{r:w} T1.$ **Atomicity**
    - T2 saw some but not all T1's writes
- $T1 \xleftarrow{r:w} T2 \xleftarrow{r:w} T1$ **Write Skew**
    - T1 and T2 each overwrote a value that the other read

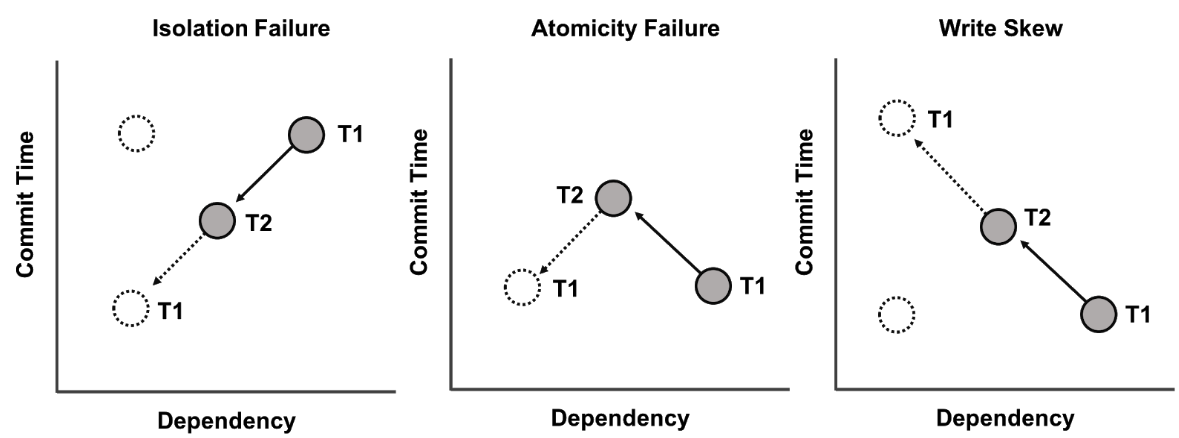


Figure B: read/write dependency (left) & read anti-dependency (right)

Isolation Levels

- Read Committed (RC)
    - Reads return the newest committed version of a record
    - Reads never block Reads
    - Writes add a new version that overwrites the latest one
    - Writes blocks only if the latter is uncommitted
- Snapshot Isolation (SI)
    - Each transaction reads from a consistent snapshot
        - newest version of each record that predates some timestamp
    - Writers must abort if they would overwrite a version created after their snapshot
- Strict Two-Phase Locking (2PL)
    - Reads return the newest version of a record, blocking if it has not committed yet
    - Writes replace the latest version, blocking if there are any in-flight reads or writes on the record by other transactions

SSN Policies (Figure C)

- **π(T)**
  - The lowest commit time of the transaction found from the back-edges

$$\pi(T) = min\left(c(U) : T \overset{b*}{\leftarrow} U\right)$$
$$= min\left(\left\{\pi(U) : T \overset{b}{\leftarrow} U\right\} \cup \{c(T)\}\right)$$

  - π(T) < c(T): The dangerous transactions committed first
  - the values of c(T) and π(T) are fixed once T has committed
    - This would be computed from only the immediate successors of a transaction in G, without traversing the whole graph

- **η(T)**
  - The commit time of the most recently committed transaction from the front-edge

$$\eta(T) = max\left(\left\{c(U) : U \overset{f}{\leftarrow} T\right\} \cup \{-\infty\}\right)$$

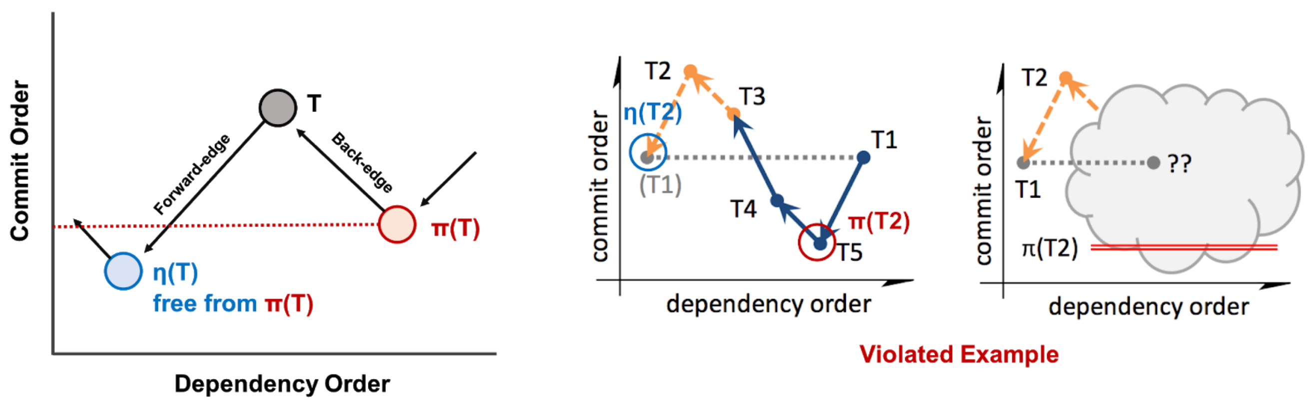
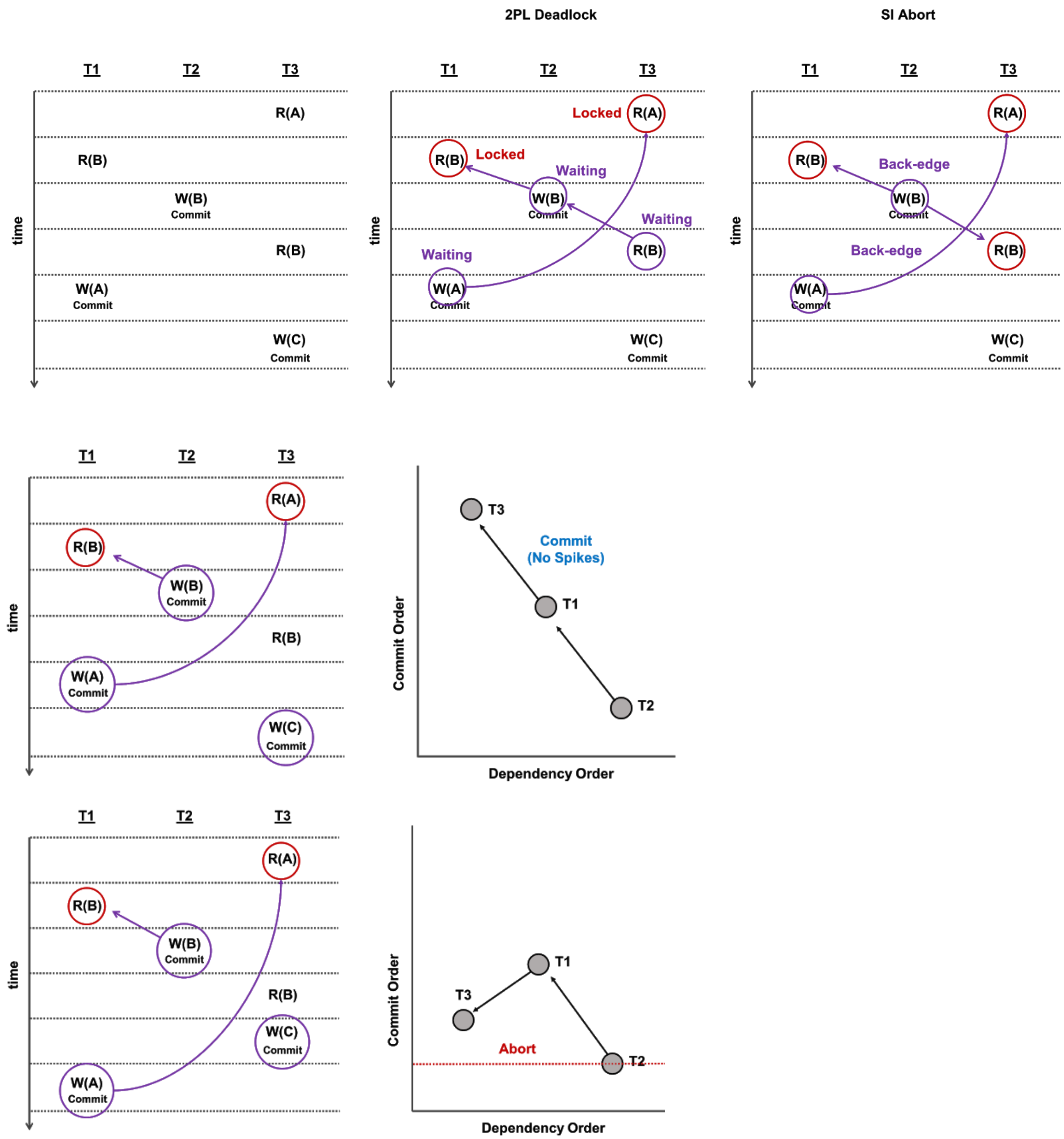


Figure C: Success example (left) & Violated example (right) T1 violates T2's exclusion window because π(T2) = c(T5) < c(T1) = η(T2). The rightmost chart depicts information that checking violation is possible with T2's local knowledge.

SSN Policies (Figure C)

- SSN ignore a large fraction of harmless back edges while still detecting all harmful ones
- Comparison among 2PL, SI, and SI + SSN



Implementation

- Read & Write sets
    - Each transaction maintain its footprints using read and write set
    - Contain all the versions read and written by the transaction
- Worker Thread
    - Walk through the version chain to find the latest committed version that is visible to the transaction

SSN read

```
# transaction t and snapshot v
# Reads the transaction t, and receives a reference to the appropriate version

def ssn_read(t, v):

    # if snapshot is not in the write_set
    if v not in t.writes:
      # update the η(T)
      # by choosing the maximum value between η(version) and the current timestamp
      t.pstamp = max(t.pstamp, v.cstamp)

    # if there is no pre π(T)
    # if the version has not yet been overwritten,
      it will be added to T's read set and checked for late-arriving overwrites
      during pre-commit
    If v.sstamp is infinity
        # add the version to the read_set
         t.reads.add(v)

    # if there is π(T)
    else:
        # compare the least value between the π(version) and π(transaction)
        t.sstamp = min(t.sstamp, v.sstamp)

     # verifies the exclusion window and aborts if a violation is detected.
     verify_exclusion_or_abort(t)
```

SSN write

```
# transaction t and snapshot v
def ssn_write(t, v):
    # if the snapshot is not in the write set
    if v not in t.writes:
```

```
    # update the η(T) by choosing
    # the maximum value between η(version) and the current timestamp
    # a write will never cause inbound read anti-dependencies
    # a write can trigger outbound read anti-dependencies
    t.pstamp = max(t.pstamp, v.prev.pstamp)

    # add snapshot to the write_set
    t.writes.add(v)

    # remove the snapshot from the read_set
    # avoid violating T's own exclusion window and trigger abort
    t.reads.discard(v)

    # verify the dependency cycle
    verify_exclusion_or_abort(t)
```

SSN commit

```
def ssn_commit(t):
    # T requests a commit timestamp c(T) in the in-flight status
    # T is no longer allowed to perform reads or writes
    t.cstamp = next_timestamp()

    t.sstamp = min(t.sstamp, t.cstamp)
    for v in t.reads:
        t.sstamp = min(t.sstamp, v.sstamp)

    for v in t.writes:
        t.pstamp = max(t.pstamp, v.prev.pstamp)


    # transactions having η(T) < π(T) are allowed to commit
    verify_exclusion_or_abort(t)
    t.status = COMMITTED

    # the transaction updates c(V) for each version it create
    # π(V) for each version it overwrote
    # (V) for each non-overwritten version it read
    for v in t.reads:
        v.pstamp = max(v.pstamp, t.cstamp)
    for v in t.writes:
        v.prev.sstamp = t.sstamp
        v.cstamp = v.pstamp = t.cstamp
```