Distributed Transaction Explained through TLA+

Accela Zhao 2018/5/23

Outline

- ***Snapshot Isolation – First Impression***
- Percolator.tla – Walkthrough
- Snapshot Isolation – Revisited
- Serializable Snapshot Isolation

Snapshot Isolation – Why this

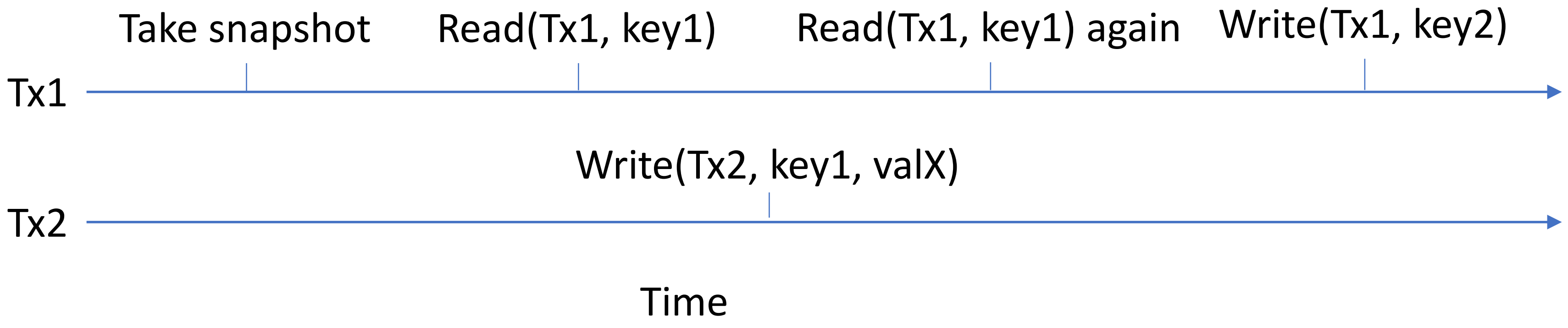
- Why talking about snapshot isolation for understanding transactions?
  - Transaction ACID
    - A – Atomicity – Usually by journaling. Or build by single row atomic operations
      - Not today's topic
    - C – Consistency – Need to manage race condition between concurrent transactions
      - Then we have Isolation Levels.
    - I – Isolation – Still, Isolation Levels.
      - Snapshot Isolation is the most commonly used Isolation Level.
    - D – Durable – Disks, replications, Paxos, erasure-coding, etc
      - Not today's topic

- As you can see, snapshot Isolation is they key to understand transaction
  - We will begin with direct impressions
  - Next we walkthrough how Percolator implements it
  - Then we can extract the accurate rules for SI to work right

P.S. "Snapshot Isolation" was mostly first proposed by Microsoft, in paper [Critique ANSI isolation levels]

Snapshot Isolation – “Snapshot”

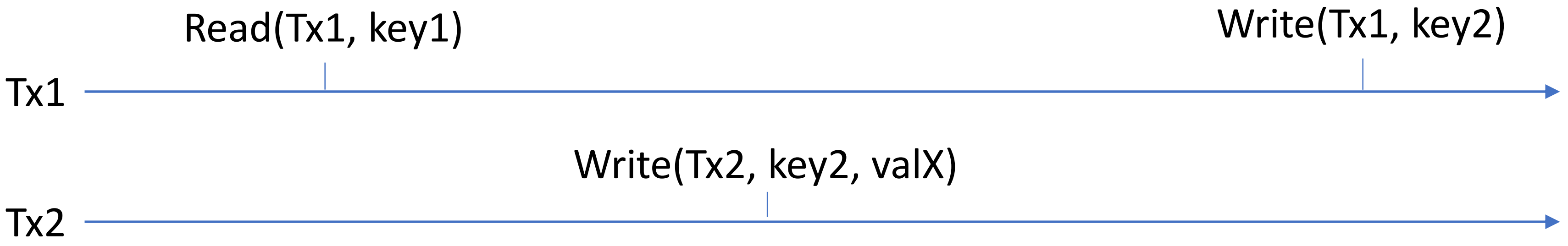
- Snapshot Isolation – Requirement 1/2 – “Snapshot”
  - Transaction (Tx) reads by first taking a “Snapshot“
  - The second read gets same value, even underlying data is changed, because we read on snapshot
  - Usually, snapshot is a timestamp. That means, Read(Tx1, key1) and Read(Tx1, key2) return values of the same time point.



Snapshot Isolation – WW-conflict

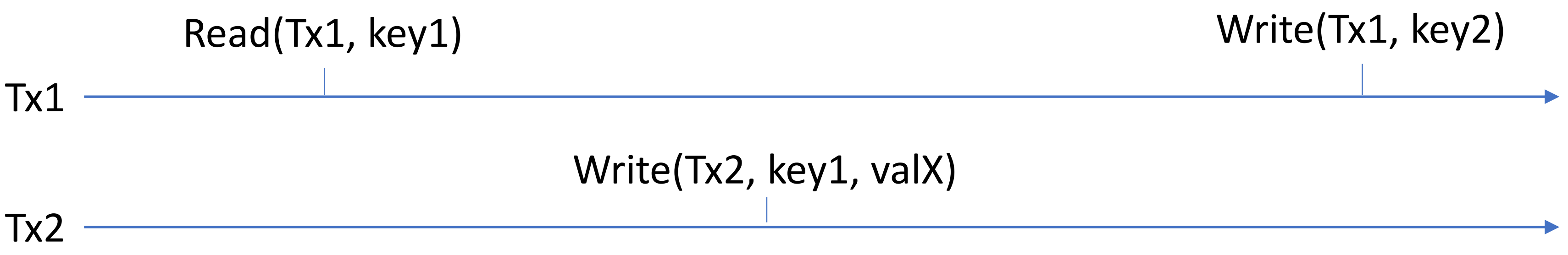
- Write-Write conflicts (ww-conflict)
  - Tx1 and Tx2 overlap. And, both Tx1 and Tx2 write to same keys.

- Snapshot Isolation – Requirement 2/2 – Abort ww-conflict
  - Actually, the rule is not a necessity for Serializability. See later (or this [Critique SI paper]).



Snapshot Isolation – RW-conflict

- Read-Write conflicts (rw-conflict)
  - Tx1 and Tx2 overlap. And, Tx2 changes what Tx1 read in the middle.
  - Tx1 is actually operating on stale data, it may result in data inconsistency.
  - Snapshot Isolation allows such case. It's called Write Skew anomaly. See SSI paper



Snapshot Isolation – Write-Skew Issue

- Example of Snapshot Isolation Write-Skew (from wiki)
  - Suppose two bank accounts V1, V2. We allow deficit, but V1 + V2 >= 0 is required.
  - V1, V2 each has \$100 balance. T1 and T2 each tries to withdraw \$200 from V1 and V2. Individually, they are OK. But in parallel, they write skew.

Bank V1	V2	T1	T2
\$100	\$100	Read V1, V2	Read V1, V2
		V1 + V2 >= \$200? => Yes	V1 + V2 >= \$200? => Yes
		Take \$200 from V1	Take \$200 from V2
		Write V1 = -\$100	Write V2 = -\$100
		Commit	Commit
-\$100	-\$100		
Inconsistency, V1 + V2 < 0			

- Some walkarounds for Write Skew (from wiki)
  - SELECT FOR UPDATE: let reads be promoted as writes, so they will conflict

Outline

- Snapshot Isolation – First Impression
- ***Percolator.tla – Walkthrough***
- Snapshot Isolation – Revisited
- Serializable Snapshot Isolation

Why TLA+ to Understand Percolator

- Who’s using TLA+
  - AWS
    - How Amazon Web Services Uses Formal Methods
    - Why Amazon Chose TLA+, Google Group
  - TiDB (Popular startup to build Spanner-like DB)
    - Github Pingcap / tla-plus
    - Official blog. An author's blog.
  - Alibaba X-DB & X-Paxos
    - InfoQ news. Reviews on Zhihu
  - Papers adopting TLA+ as format proof
    - CASPaxos
  - Lamport is putting significant effort on TLA+
    - Lamport publications. See how many "TLA"s

- TLA+ Benefits
  - Strict math, complete, concise
    - Good for understanding complex protocols like Percolator
  - Auto tools
    - TLC – Check state enumerating and invariants
    - TLAPS – Math derive the invariants

- Learning TLA+
  - Lamport's TLA+ page, the TLA+ book
    - Part I is mostly what we need
  - Links in “Who’s using TLA+”
  - Github DrTLAPlus

What is Percolator?

- Google's distributed transaction implementation, for batch web index processing, built on BigTable
  - Paper: Large-scale Incremental Processing Using Distributed Transactions and Notifications

- Achieves ACID transaction, with Snapshot Isolation, with MVCC and optimistic locking, and falls in category of 2-phase locking
  - We'll see how these "words" come from later

- A popular distributed transaction implementation
  - TiDB is borrowing a lot from Percolator. CockroachDB is also learning from it
  - Spanner share many things similar to Percolator
    - It can go to another topic

- Github tla-plus / Percolator / Percolator.tla TLA+ spec
  - Good for understanding. And can tweak/run with TLC.

Percolator.tla - State Overview

- Start – Obtain start timestamp
  - Obtain Tx’s start timestamp `start_ts` from a central timestamp oracle

- Get – Will do many things
  - Cleanup stale locks
    - If a lock is older than me, clean it. It will make former Tx unable to commit (i.e. new Tx preempts old).
      - Paper shows more graceful conditions of cleaning lock
  - Commit secondary keys
    - A Tx write many keys, Percolator select one as primary key, others as secondary
    - Secondary keys are lazy committed by other Tx’s Get.
  - Doing actual read

```
ClientOp(c) ==
    \/ Start(c)
    \/ Get(c)
    \/ Prewrite(c)
    \/ Commit(c)
    \/ Abort(c)
```

Percolator.tla - State Overview

- Prewrite – Lock every key before commit
  - "Lock" in Percolator is quite different from other systems
    - Just a DB record. No actual pending.
    - If the key has newer write than me, cannot lock.
    - Acquire lock will *write* data (i.e. bal:data)
- Commit
  - Write "write record" (i.e. bal:write), which makes data visible, and release locks.
  - Tx only commits primary key, other secondary keys are left lazy commit, by following Txs (see Get)

```
ClientOp(c) ==
    \/ Start(c)
    \/ Get(c)
    \/ Prewrite(c)
    \/ Commit(c)
    \/ Abort(c)
```

Percolator data structure in BigTable

key	*bal:data*	*bal:lock*	*bal:write*
Bob	*6:* *5:* $10	*6:* *5:*	*6:* data @ 5 *5:*
Joe	*6:* *5:* $2	*6:* *5:*	*6:* data @ 5 *5:*

Version: Actual Data

Version: Pointer to data.
Only when pointed by this column, data is considered committed

Percolator.tla – Walkthrough the Spec

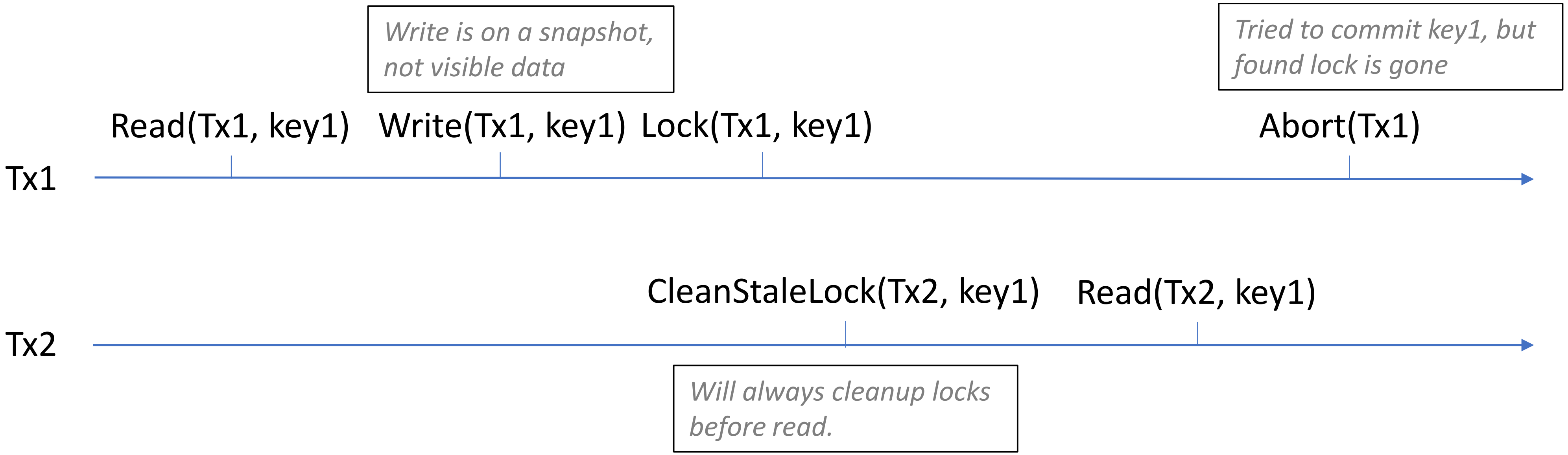
- Github [tla-plus / Percolator / Percolator.tla] TLA+ spec
  - (Planned to walkthrough with previous slides)

- Some hints for understanding TLA+ symbols
  - “/\” means “AND”, “\/” means “OR”. (They are math)
  - “key_data' = [key_data EXCEPT ![l.primary] = @ \ {[ts |-> l.ts]}]”
    - Means “key_data[l.primary] removes [ts: l.ts]”

- Demo – Run Percolator.tla with TLC
  - `java -cp ./tla2tools.jar tlc2.TLC -deadlock -workers 4 Test1`

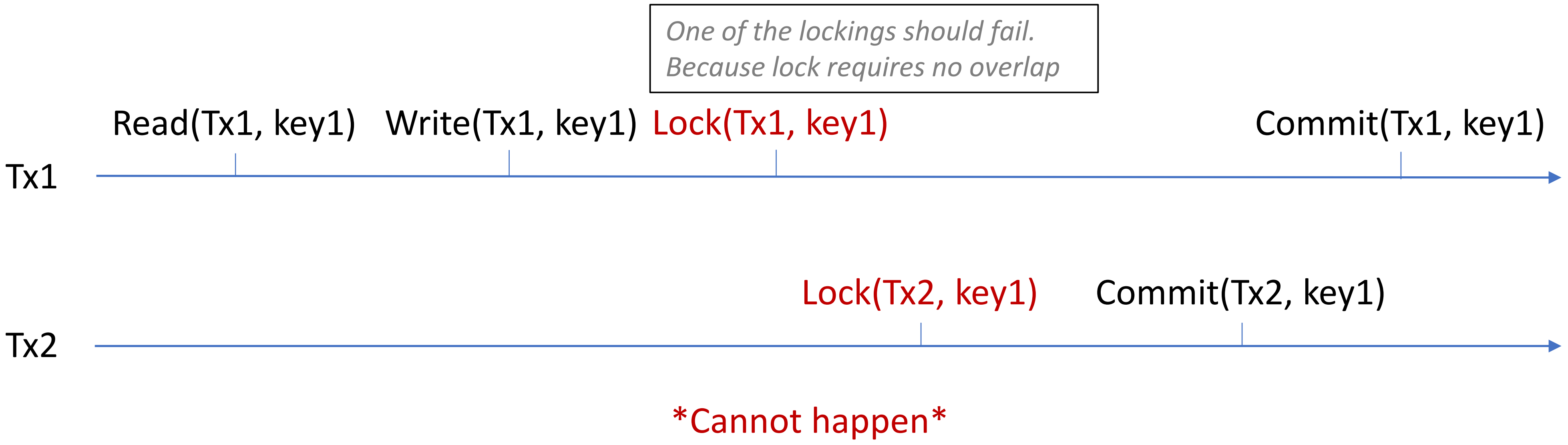
Percolator.tla - Examples

- Tx1 is preempted by Tx2
  - Although Tx1 acquired lock, the lock is later cleaned up by Tx2
  - Tx1 cannot commit.



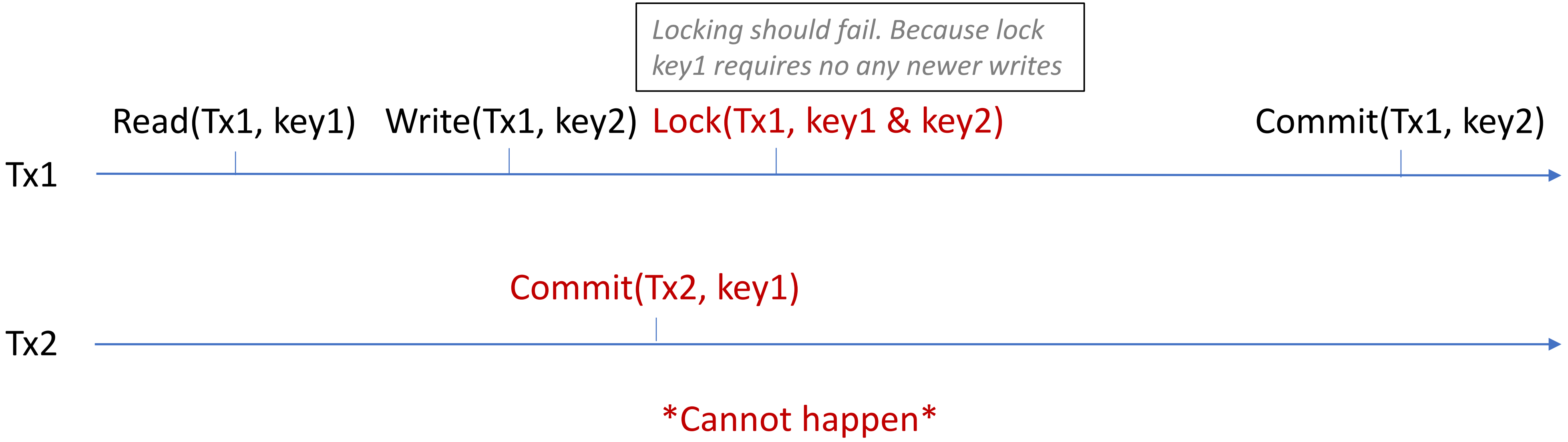
Percolator.tla - Examples

- WW-conflict is aborted
  - If both Tx1 and Tx2 tries to commit to same key, their locks overlaps
  - One of Tx1 or Tx2 will abort.



Percolator.tla - Examples

- RW-conflict is aborted
  - Tx1 locks key1. Locking requires key1 has no any newer writes.
  - Since key1 was modified in the middle, Tx1 cannot lock it and will abort.



Outline

- Snapshot Isolation – First Impression
- Percolator.tla – Walkthrough
- ***Snapshot Isolation – Revisited***
- Serializable Snapshot Isolation

How Percolator.tla Enforces Snapshot Isolation

- Snapshot Isolation – Requirement 1/2 – “Snapshot”
  - Read/write is based on timestamp.
  - If a key is read, then older commit cannot proceed
    - Otherwise, visible history would have been changed
    - Enforced by: In Get, newer read will clean all stale lock. Older Tx cannot commit without lock

Enforced by

```
readKey(c) ==
  LET
    start_ts == client_ts[c].start_ts
    primary == client_key[c].primary
    secondary == client_key[c].secondary
  IN
    \E k \in {primary} \union secondary :
      /\ ~hasStaleLock(k, start_ts)
      /\ key_last_read_ts[k] < start_ts
      /\ key_last_read_ts' = [key_last_read_ts EXCEPT ![k] = start_ts]
      /\ UNCHANGED <<key_data, key_lock, key_write, key_si>>
```

Verified by

```
checkSnapshotIsolation(k, commit_ts) ==
  IF key_last_read_ts[k] >= commit_ts
  THEN
    key_si' = [key_si EXCEPT ![k] = FALSE]
  ELSE
    UNCHANGED <<key_si>>
```

How Percolator.tla Enforces Snapshot Isolation

- Snapshot Isolation – Requirement 2/2 – Abort "ww-conflict"
  - All primary and secondary keys, no matter read or write, all locked before commit
  - Only one overlapped lock can succeed
  - Lock enforces "no any newer writes"

Enforced by

```
canLockKey(k, ts) ==
  LET
    writes == {w \in DOMAIN key_write[k] : key_write[k][w].ts >= ts}
  IN
    /\ key_lock[k] = {}  \* no any lock for the key.
    /\ writes = {}       \* no any newer write.
```

Verified by

WriteConsistency == …
LockConsistency == …
CommittedConsistency == …

- P.S. I think rw-conflict abort is also enforced in Percolator.tla
  - Because all read/write keys are locked. And lock requires "no any newer writes".
  - And, this ensures Serializability
    - Critique SI paper proves "rw-conflict avoidance is sufficient for Serializability."

How Percolator Achieves …

- ACID
  - A – No journal, but BigTable provides atomic row operation
    - And, during commit, Percolator only commits primary key.
  - C / I – The snapshot Isolation as illustrated previously
  - D – Data & transaction states in BigTable.
- MVCC
  - Percolator provides multi-version with timestamp
  - Concurrency control is based on timestamp & locking
- Optimistic locking - Likely
  - Read never block (actually preempt former Tx)
  - Tx will executed first, without waiting for locks, but under the risk of abort
- Falls in 2-Phase Locking category
  - We still see the Prewrite step first prepare each key with locking. Then we commit

Thinking in Abstract Level

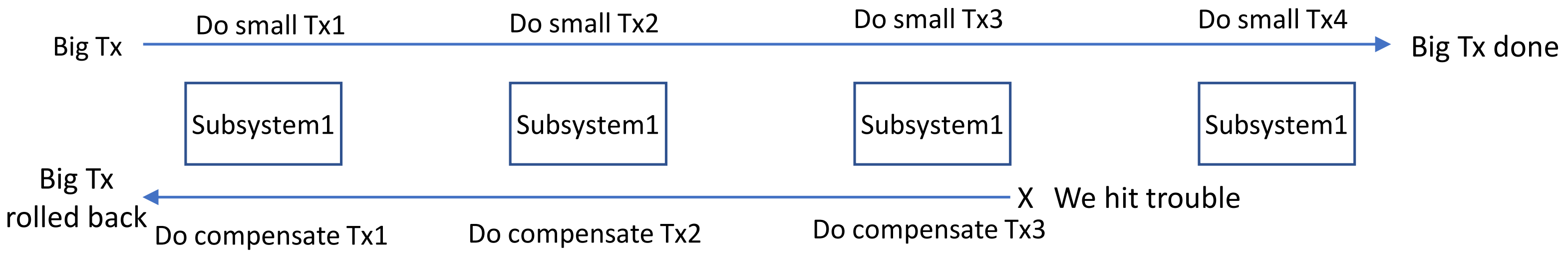
- What is the essence of Snapshot Isolation?
  - Reads never lock. That's why it's faster
  - To abort ww-conflict, we still needs locking
    - Approach 1: let newer Tx wait
      - Then we are using traditional locks
    - Approach 2: abort newer Tx
      - Then we have something like optimistic read – compare – succ or abort
    - Approach 3: abort older Tx
      - What Percolator does. New Tx cleans locks of older Tx.

Thinking in Abstract Level

- How are we implementing the transaction?
  - Problem 1: We need concurrency control for transactions
    - Approach 1: we use timestamp
      - Then we go to approaches of Percolator, etc MVCC
    - Approach 2: we use locks
      - Strict-2PC locking is still the traditional way to enforce Serializability
  - Problem 2: How do we enforce the ordering of transaction read/writes?
    - Approach 1: in distributed manner
      - Approach 1.1: with timestamps
        - As we see in Percolator, careful arranging locks and timestamp compares
      - Approach 1.2: with locks
        - Tx pending on locks, so they are ordered. Traditional implementation.
    - Approach 2: centralized coordinator
      - Critique SI paper is using a centralized status oracle, to control the total ordering
      - Calvin Transaction paper is using a scheduler, which knows all transactions

Thinking in the Abstract Level

- Jump out of the box? - Eventual consistency transaction + compensations
  - Distributed transactions of weaker than ACID. But quite useful and popular at Internet companies.
  - Background
    - We have many subsystems. Each subsystem supports ACID transaction individually.
    - But we lack cross-subsystem big transactions.
  - How it works
    - Split big transaction into small ones, to be executed on each subsystem.
    - Carry out small transactions one by one in a known workflow.
      - I.e. Weak consistency, but propagating in a controlled order
    - Eventually all small transactions finish. Then big transaction is done.
  - How to rollback
    - If we cannot proceed at certain step, e.g. conflict, we start rollback
    - Rollback by compensation. I.e. use another transaction to “fix” things back.



Outline

- Snapshot Isolation – First Impression
- Percolator.tla – Walkthrough
- Snapshot Isolation – Revisited
- ***Serializable Snapshot Isolation*** *(Quick Look)*

Why Serializable Snapshot Isolation – Quick Look

- What Serializable Snapshot Isolation (SSI) can do?
  - Serializable isolation level.
  - No need for 2PC. Performance is acceptable.
    - Previously, Serializable level needs 2PC.
    - Even in Percolator.tla, you can see it locks all keys.
  - Can be built on Snapshot Isolation. Less engineering effort.

- How does SSI do it?
  - Rw-conflict abort can get Serializability. But, it falsefully aborts unnecessary transactions, which are Serializable however.
  - Theorem: Only needs to abort the “dangerous structure”, i.e. graphs with two consecutive rw-dependency edges.
    - Rw-conflict aborting, however, aborts on every single such edge

- Papers
  - SSI proposed in this paper
  - PostgreSQL implements SSI and illustrates it well

References

- A Critique of ANSI SQL Isolation Levels
  - Proposed "Snapshot Isolation"
- A Critique of Snapshot Isolation
  - Explain Snapshot Isolation well
- Calvin: Fast Distributed Transactions for Partitioned Database Systems
  - Another distributed transaction implementation
- weidagang/distributed_mvcc_cross_row_transaction.py
  - Python implemented Percolator protocol
- Serializable Isolation for Snapshot Databases
  - Proposed "Serializable Snapshot Isolation"
- Serializable Snapshot Isolation in PostgreSQL
  - Explain SSI well, and implementation details
- Compensating Transactions: When ACID is too much
  - Eventual consistency distributed transaction
- TiDB Transaction Model. CockroachDB transaction Model. Hacker News discussions.
  - They are popular opensource distributed SQL databases.