Concurrency control

(OCC and MVCC)

COS 518: *Advanced Computer Systems*
Lecture 6

Michael Freedman

Q: What if access patterns rarely, if ever, conflict?

Be optimistic!

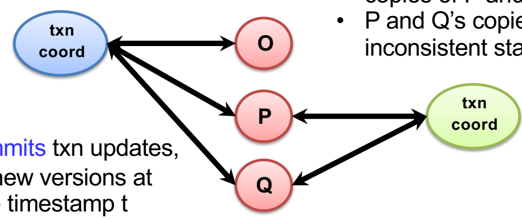
- Goal: Low overhead for non-conflicting txns
- Assume success!
  - Process transaction as if would succeed
  - Check for serializability only at commit time
  - If fails, abort transaction
- **Optimistic Concurrency Control (OCC)**
  - Higher performance when few conflicts vs. locking
  - Lower performance when many conflicts vs. locking

OCC: Three-phase approach

- **Begin:** Record timestamp marking the transaction's beginning
- **Modify** phase:
  - Txn can read values of committed data items
  - Updates only to local copies (versions) of items (in db cache)
- **Validate** phase
- **Commit** phase
  - If validates, transaction's updates applied to DB
  - Otherwise, transaction restarted
  - Care must be taken to avoid "TOCTTOU" issues

OCC: Why validation is necessary

- New txn creates shadow copies of P and Q
- P and Q's copies at inconsistent state



When commits txn updates, create new versions at some timestamp t

OCC: Validate Phase

- Transaction is about to commit.
  System must ensure:
  - Initial consistency: Versions of accessed objects at start consistent
  - No conflicting concurrency: No other txn has committed an operation at object that conflicts with one of this txn's invocations

OCC: Validate Phase

- Validation needed by transaction T to commit:
- For all other txns O either **committed** or **in validation** phase, one of following holds:
  A. O completes commit before T starts modify
  B. T starts commit after O completes commit, and ReadSet T and WriteSet O are disjoint
  C. Both ReadSet T and WriteSet T are disjoint from WriteSet O, and O completes modify phase.
- When validating T, first check (A), then (B), then (C). If all fail, validation fails and T aborted

2PL & OCC = strict serialization

- Provides semantics as if only one transaction was running on DB at time, in serial order

  \+ Real-time guarantees

- 2PL: Pessimistically get all the locks first
- OCC: Optimistically create copies, but then recheck all read + written items before commit

2PL & OCC = strict serialization

- Provides semantics as if only one transaction was running on DB at time, in serial order

  + Real-time guarantees

- 2PL: Pessimistically get all the locks first
- OCC: Optimistically create copies, but then recheck all read + written items before commit

9

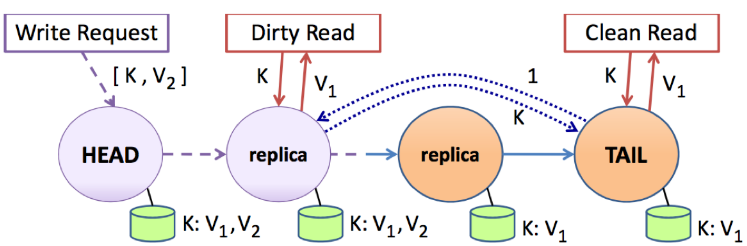
Multi-version concurrency control

Generalize use of multiple versions of objects

10

Multi-version concurrency control

- Maintain multiple versions of objects, each with own timestamp. Allocate correct version to reads.
- Prior example of MVCC:



11

Multi-version concurrency control

- Maintain multiple versions of objects, each with own timestamp. Allocate correct version to reads.
- Unlike 2PL/OCC, reads never rejected
- Occasionally run garbage collection to clean up

12

MVCC Intuition

- Split transaction into read set and write set
  - All reads execute as if one "snapshot"
  - All writes execute as if one later "snapshot"
- Yields snapshot isolation < serializability

Serializability vs. Snapshot isolation

- Intuition: Bag of marbles: ½ white, ½ black
- Transactions:
  - T1: Change all white marbles to black marbles
  - T2: Change all black marbles to white marbles
- Serializability (2PL, OCC)
  - T1 → T2 or T2 → T1
  - In either case, bag is either ALL white or ALL black
- Snapshot isolation (MVCC)
  - T1 → T2 or T2 → T1 or T1 || T2
  - Bag is ALL white, ALL black, or ½ white ½ black

Timestamps in MVCC

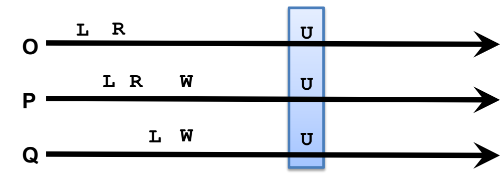
- Transactions are assigned timestamps, which may get assigned to objects those txns read/write
- Every object version O_V has both read and write TS
  - ReadTS: Largest timestamp of txn that reads O_V
  - WriteTS: Timestamp of txn that wrote O_V

Executing transaction T in MVCC

- Find version of object O to read:
  - # Determine the last version written before read snapshot time
  - Find O_V s.t. max { WriteTS(O_V) | WriteTS(O_V) <= TS(T) }
  - ReadTS(O_V) = max(TS(T), ReadTS(O_V))
  - Return O_V to T
- Perform write of object O or abort if conflicting:
  - Find O_V s.t. max { WriteTS(O_V) | WriteTS(O_V) <= TS(T) }
  - # Abort if another T' exists and has read O after T
  - If ReadTS(O_V) > TS(T)
    - Abort and roll-back T
  - Else
    - Create new version O_W
    - Set ReadTS(O_W) = WriteTS(O_W) = TS(T)

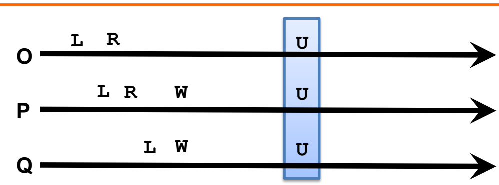
Distributed Transactions

Consider partitioned data over servers



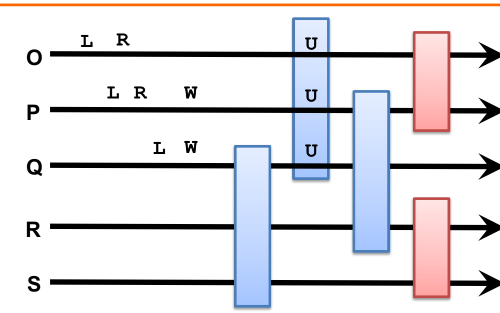
- Why not just use 2PL?
  - Grab locks over entire read and write set
  - Perform writes
  - Release locks (at commit time)

Consider partitioned data over servers



- How do you get serializability?
  - On single machine, single COMMIT op in the WAL
  - In distributed setting, assign global timestamp to txn (at sometime after lock acquisition and before commit)
    - Centralized txn manager
    - Distributed consensus on timestamp (not all ops)

Strawman: Consensus per txn group?



- Single Lamport clock, consensus per group?
  - Linearizability composes!
  - But doesn't solve concurrent, non-overlapping txn problem