EECS 491
Introduction to Distributed Systems

Fall 2019

Harsha V. Madhyastha
Distributed Transactions

- Partitioning of state necessary to scale
- Partitioning results in the need for transactions
  - Atomically execute operations across shards

- Examples:
  - Add meeting to calendars of two participants
  - Transfer money from one account to another
  - Looking up balance of two accounts
Concurrency + Serializability

- Execution of transactions:
  \[ T_1 \quad T_2 \quad T_3 \]

- Externally visible effects:
  \[ T_2 \quad T_1 \quad T_3 \]
Example of Serializability

- Concurrent execution of transactions:
  - T1: Transfer $10 from Alice to Bob
  - T2: Read balance in Alice’s and Bob’s accounts
  - Initial balance of $100 in both accounts

- Permissible outputs for T2:
  - (Alice: $100, Bob: $100) or (Alice: $90, Bob: $110)

- Invalid outputs for T2:
  - (Alice: $90, Bob: $100) or (Alice: $100, Bob: $110)
Achiving Serializability

- How to execute distributed transactions?

- Goals:
  - Ensure externally visible effects are serializable
  - Maximize concurrency
Example Scenario

- Many students are each looking to host a party
- Each host invites a subset of other students
- Party is on only if all invitees can make it

- Consensus among all students unnecessary
  - Multiple parties ok if disjoint set of invitees
Strawman Solution

ScheduleParty(time slot, list of invitees)

1. Attempt to add entry to calendar of all invitees
2. If success, done
3. If any calendar update fails, delete all updates and retry step 1

Problem: Between steps 1 and 3, a user might be shown as busy, when they eventually won’t

How to avoid this?
Achieving Serializability

- One big lock → Fine-grained locks
- Transaction coordinator (TC) acquires locks on all data involved
- Once locks acquired, execute transaction and release locks
Two Phase Locking

TC

P1

P2

P3

Lock

Commit
Two Phase Locking

TC

Lock

Abort

P1

P2

P3
Two Phase Locking

- TC acquires locks on all necessary shards before attempting transaction
- Disjoint transactions can execute concurrently
- Related transactions wait on each other
Can we reduce latency for read-only transactions?

Return data in first round if lock not held
Optimizing Read-Only Txns

- Concurrent execution of transactions:
  - T1: Transfer $10 from Alice to Bob
  - T2: Read balance in Alice’s and Bob’s accounts
  - Initial balance of $100 in both accounts

- Problem if, when TC2 is attempting to acquire locks, it reads data directly if lock not held?
  - TC2 reads Alice’s account balance as $100
  - TC1 acquires locks, transfers $100, releases locks
  - TC2 reads Bob’s account balance as $110
Lock-free Read-Only Txns

TC1

P1

P2

TC2

Read

Lock

Commit
Fault Tolerance of 2PL

TC1

P1

P2

TC2

Lock

Commit

Lock

Commit

October 29, 2019
EECS 491 – Lecture 16
Fault Tolerance of 2PL

Okay to commit after timeout?

TC

P1

P2

P3

Lock

Commit
Fault Tolerance of 2PL

TC

P1

P2

P3

Lock

Abort
Fault Tolerance of 2PL

When can we garbage collect transaction log?
Garbage Collection

- When can we garbage collect each of these?
- Lock acquisition in node log:
  - Node receives commit from TC and writes to its log
- Commit operation in TC log:
  - After all nodes say transaction committed
- Commit operation in node log:
  - Upon applying operation to local state
Transaction cannot succeed if any partition unavailable

Every partition should be implemented as an RSM
Fault Tolerance

RSM to offer high availability, but equivalent to single server

Partition RSM state in order to scale

Implement each partition as an RSM to improve availability of cross-partition transactions
Upcoming Schedule

- This week
  - Thursday: Project 3 due
  - Friday: Review of material for midterm in discussion

- Next week
  - Tuesday: Solutions to sample midterm exam
  - Thursday:
    » No lecture
    » Midterm exam 6:30-10:30pm
  - Friday: No discussion section
Improving Concurrency of Transactions

- System stores employee → salary mappings
- T1: TotalSalary = sum of employee salaries
- T2: MedianSalary = median employee salary
- Both T1 and T2 require locks for all employees
  - → T1 and T2 must run one after the other
- But … no conflict if T1 and T2 run concurrently
- Acquire locks only for data txn will modify?
- T3: Increase all employee salaries by 10%
Managing Concurrency

- Two-phase locking is pessimistic
  - Acquire locks assuming conflict will occur

- Optimistic concurrency control
  - Execute transaction assuming no conflict
  - Verify that no conflict before commit
Optimistic Concurrency Control

- **Read** required data and compute results
  - TC ideally reads from local cache
- **Verify** that no conflicts
  - TC asks relevant partitions whether safe to commit
- **Commit** results

---

**Diagram:**

- **Read Phase**
- **Validate Phase**
- **Write Phase**
Centralized Validation

- TC fetches required data and executes transaction locally
- TC sends transaction to validation server
- Validation server evaluates if commit → conflict
  - If no, ask TC to commit results
  - If yes, ask TC to re-read and re-execute transaction
If x=0, y=0, and z=0 initially, how to evaluate a set of transactions whether safe to execute concurrently?