Transactions

Intel (TX memory): Transactional Synchronization Extensions (TSX)

Goal – A Distributed Transaction

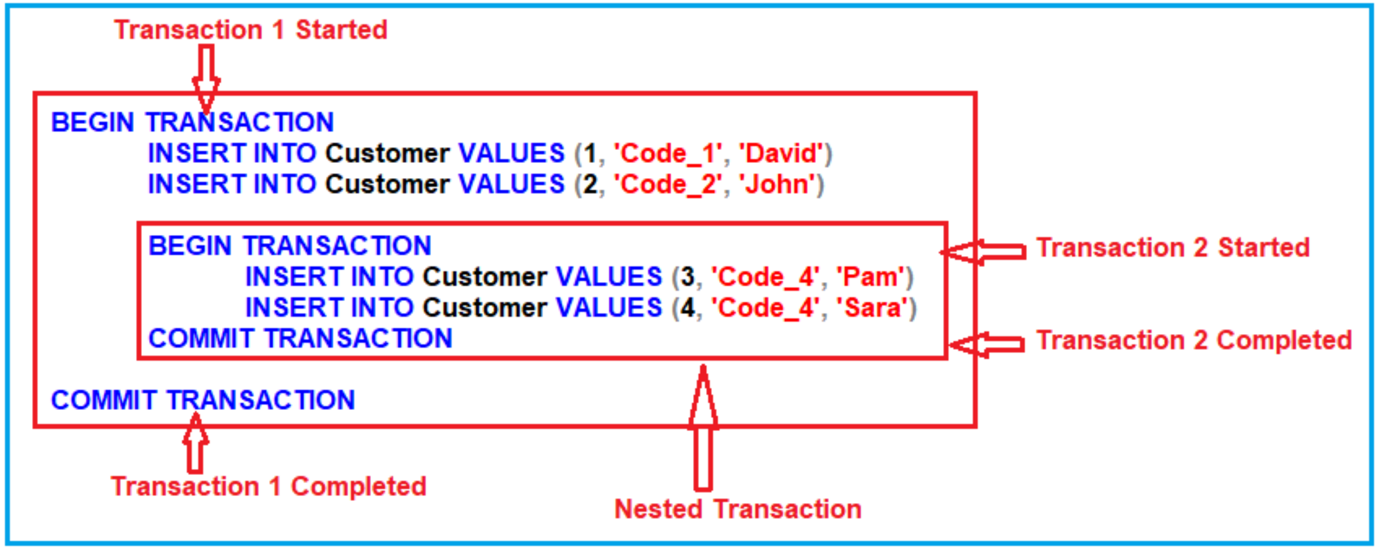
- We want a transaction that involves multiple nodes
- Review of transactions and their properties
- Things we need to implement transactions
  - Locks
  - Achieving atomicity through logging
    - Roll ahead, roll back, write ahead logging
- Finally, 2 Phase Commit (aka 2PC) and 3PC
- Lead into Paxos

Transactions - Definition

- A transaction is a sequence of data operations with the following properties:
  - A Atomic
    - All or nothing
  - C Consistent
    - Consistent state in => consistent state out
  - I Independent
    - Partial results are not visible to concurrent transactions
  - D Durable
    - Once completed, new state survives crashes

Transactional API

- Interface
  - tran = TranMonitor.**begin** ()
    - Do some stuff within a transaction session
  - tran.**commit**()
  - tran.**abort**()



Serializability

- A set of transactions is serializable iff
  - resulting state is equivalent to that produced by some serial ordering of those transactions
- They don't actually have to run in serial order
  - system just ensures that actual outcome is the same as if they had

Importance of independence

- Possible problems if we don't have it
  - lost update
    - t1 and t2 read *x* and then write *x*, t1's update is lost
  - inconsistent retrieval
    - Intermediate state may be inconsistent (e.g., sum=x+y violated)
  - dirty read
    - t1 updates *x*, t2 reads *x*, t1 aborts; t2 has dirty value of *x*
  - premature write
    - t1 update x, t2 update *x*, t1 aborts, t2's update (to x) is lost

Importance of independence

* lost update
  * t1 and t2 read *x* and then write *x*, t1's update is lost

  Example:

* One transaction may overwrite the result of another.
* Example: Transaction T wants to increase b's balance by 10%, transferring from a.
  * T1: bal = b.getBalance()
  * T2: b.setBalance(bal*1.1)
  * T3: a.withdraw(bal/10)
* Transaction U wants to increase b's balance by 10%, transferring from c.
  * U1: bal = b.getBalance()
  * U2: b.setBalance(bal*1.1)
  * U3: c.withdraw(bal/10)
* Problem: suppose order is T1, U1, U2, T2, T3, U3.

Importance of independence

* premature write
  - t1 update x, t2 update *x*, t1 aborts, t2's update is lost

  Example:
  - a - balance is $100
  - T: a.setBalance($105) - (before image: 100)
  - U: a.setBalance($110) - (before image: 105)
  - U commits, T aborts and resets to 100 -- should be 110
  - If T aborts then U aborts, result will be 105, but should be 100.

Two Possible (pessimistic) Approaches

- Two Phase Locking
- Strict Two Phase Locking

Two Phase Locking

- Locks
  - * reader/writer locks
  - * acquired **as** transaction proceeds
  - * no more acquires after first release
- Phase 1
  - acquire locks and access data, but release no locks
- Phase 2
  - access data, release locks, but acquire no new locks
  - commit/abort transaction at end

Q Semantics of two-phase locking

- Does the Two-Phase Locking protocol ensure
  - serializability?
  - independence?

- How?

Semantics of two-phase locking

- Ensures serializability
  - if transactions have no conflicting lock access
    - order arbitrarily
  - for any transactions with conflicting lock access
    - order transactions based on order lock is acquired
  - transactions are serialized
    - because, no lock is acquired after first release
    - deadlocks are still possible
- Does not ensure independence
  - we still have premature write and dirty read problems
  - E.g., t1 releases x, t2 acquires x, then t1 aborts

Strict two phase locking

- Like two-phase locking, but
  - release no locks until transaction commits
- Phase 1:
  - acquire locks and access data, but release no locks
- Phase 2:
  - Commit/abort transaction and then release all locks
- Ensures both serializability and independence