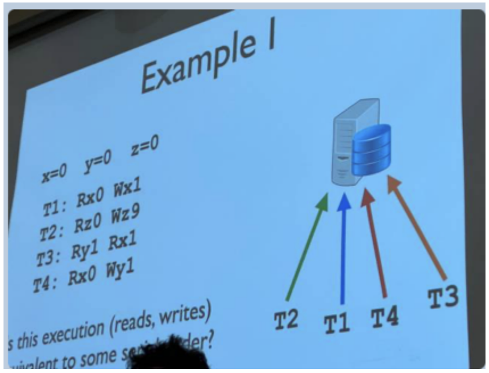
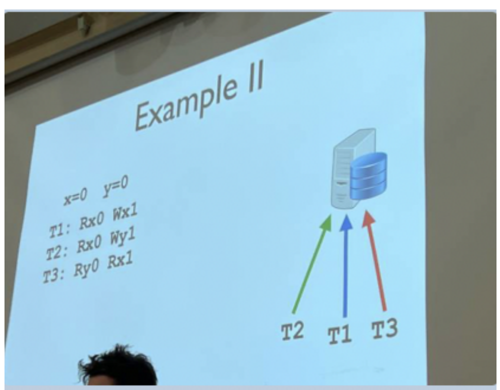
CAS CS 350
Lec 16

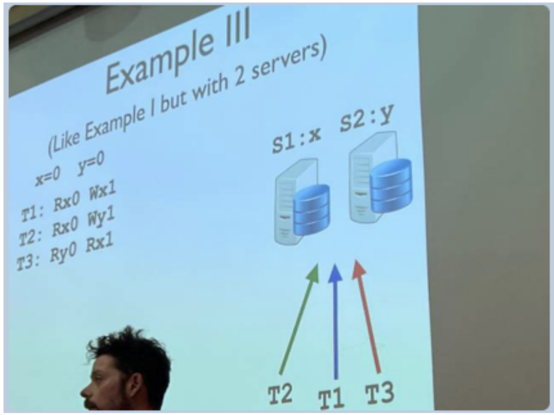
Transactions

1. Overview
    a. Protocol (all or nothing) → atomicity property (if one of the servers commit, everybody commits), (if one of the servers abort, everybody aborts)
2. Concurrency
    a. What about concurrent transactions?
        i. We usually want concurrency control as well as atomic commit
        ii. E.g. Transfer along with audit – sum
    b. TA1: add(X,1) add(Y,-1)
       TA2: tmp1 = get(X) tmp2 = get(Y) print tmp1,tmp2
       If TA2 runs concurrently it is possible to make money eg. not see subtraction but see addition
3. Serializability
    a. Not to be confused with linearizability
    b. A correctness condition for transactions, i.e. sequences of R/W operations on one or more objects
    c. Guarantee: the concurrent execution of transactions equivalent to a serial execution
    d. Each transaction is executed as a whole one after the other
        i. If no equivalent serial execution exists, a transaction have read results of other incompatible transactions
        ii. If the hypothetical order respects the real time the transactions were committed by the client, then
            1. Strict serializability → extra real time guarantee
4. Linearizability
    a. Strong consistency model
    b. A correctness condition for R/W operations
    c. Guarantee: Read on object X will see the value of the latest completed write on X
    d. Raft supports linearizable semantics, i.e, clients cannot see stale data
    e. Raft is linearizability → everything goes through the leader and leader decides what is committed or not

5. Example 1
    a. 

    b. RX0 means that transaction T1 read variable x as 0
    c. WX1 means that transaction T1 set variable x to 1
    d. Works concurrently
    e. Is this execution equivalent to some serial order?
        i. Yes: T4, T1, T3, T2 (T2 can go anywhere)
        ii. T2 → variable Z is not used in the transaction, which means that it can go anywhere
    f. Serial → transactions appear to have occurred in some total order, and if not, it could lead to inconsistency
6. Example 2
    a. 

    b. Is this execution equivalent to some serial order?
        i. No
        ii. T1, T3 required (via x)
        iii. T3, T2: required (via y)
        iv. But T2, T1 required (via x)
        v. Circular Dependency
    c. This means that in practice, it means that it got data from another place, which could lead to inconsistency

7. Example 3
    a. 

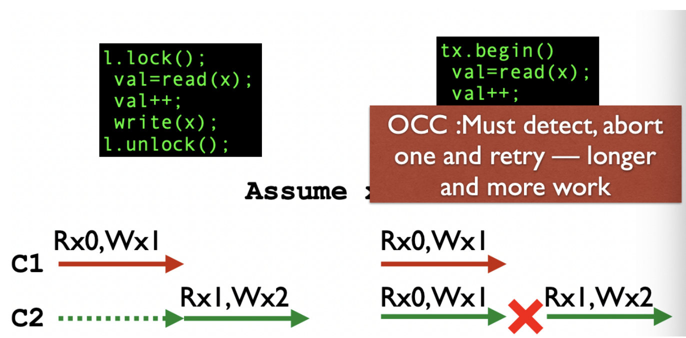
    b. (like example 1 but with 2 servers)
    c. Part of T2, the first reads X, is run by server 1 and part of T2, the write Y, is run by server 2
    d. S1: x only (all operations on X is run by server 1)
        i. T1: Rx0 Wx1
        ii. T2: Rx0
        iii. T3: Rx1
    e. There is serializability based on x → T2, T1, T3
    f. S2: y only (all operations on Y is run by server 2)
        i. T2: Wy1
        ii. T3: Ry0
    g. There is serializability based on y → T3, T2
    h. Problem: Two different orders for T3 and T2 → inconsistency since we are interested in finding global order
    i. How could we identify such inconsistency?
        i. Send the transaction order for each server to the transaction coordinator and the coordinator checks whether there is a conflict → there may be fault tolerance (if one transaction does not matter of the order it goes)
            1. Have coordinator tell servers the order they need to execute the transactions (not most efficient)
            2. Coordinator impose global order to servers → assign each transaction a timestamp or sequence number (servers should expect these sequence numbers)
8. Resolving global order
    a. Each server can send possible serial order to the transaction coordinator to decide
    b. We can assign timestamps (or sequence numbers) to transactions so that the servers find a serial order that respects the timestamp order

9. Example 4
    a. Same as Example 3 but with timestamps (can also use sequence numbers)
    b. T1@100: Rx0 Wx1
       T2@110: Rx0 Wy1
       T3@105: Ry0 Rx1
    c. The sequence numbers define a global order of transactions (can be order that transaction arrives at the coordinator)
    d. Ts impose T1, T3, T2 (invalid) for S1 (x only)
    e. Ts impose T3, T2 (valid) for S2 (y only)
    f. The set of transactions is not serializable
10. Concurrency Control (Optimistic vs pessimistic)
    a. Enforce serializability in practice

```
l.lock();
 val=read(x);
 val++;
 write(x);
l.unlock();
```

```
tx.begin()
 val=read(x);
 val++;
 write(x);
tx.end();
```

    b.
    c. Pessimistic → each transaction locks every variables and run
    d. Optimistic → system lets transactions run → checks whether there is a conflict afterwards
    e. Difference between two
    f. Assume x = 0
    g.




    h. It is possible that both threads can give same results
    i. Optimistic → faster but can be wrong → need to take extra steps
    j. Optimistic → must detect, abort one and retry – longer and more work
    k. No clear winner (exist tradeoff between two options)
        i. If there are many dependencies and conflicts → pessimistic is a better approach
    l. Optimistic is better if few conflicts, otherwise overheads could be larger than lock waits in case of conflicts

11. Two phase locking
    a. A pessimistic approach to enforce serializability
    b. Each use of a record automatically waits for and acquires the record's lock
    c. Thus add() handler implicitly acquires lock when it uses record x or y
    d. Locks are held until "after" commit or abort
    e. Why hold locks until after commit/abort?
        i. Locks here support serializability → enforce order
        ii. When transactions conflict, locks delay one to force serial execution
        iii. When transactions don't conflict, locks allow fast parallel execution