Cloud computing and distributed systems

Transactions and concurrency control

Zeynep Yücel

Ca' Foscari University of Venice
zeynep.yucel@unive.it
yucelzeynep.github.io

Introduction

- Transaction Definition: set of operations treated as as an indivisible unit.
  - Either complete a transaction or fully erase it.
- Goal of transactions: Keep a consistent state of objects during
  - Multiple transactions
  - Server crash
- Recoverable Objects: are those that can be recovered after the server crashes.
  - Stored in volatile or persistent memory.

Introduction

Operations of the *Account* interface

deposit(amount)
deposit *amount* in the account

withdraw(amount)
withdraw *amount* from the account

getBalance() → *amount*
return the balance of the account

setBalance(amount)
set the balance of the account to *amount*

Operations of the *Branch* interface

create(name) → *account*
create a new account with a given name

lookUp(name) → *account*
return a reference to the account with the given name

branchTotal() → *amount*
return the total of all the balances at the branch

- Each account is a remote object whose interface, *Account*, provides deposit, withdraw and get/setBalance operations.
- Each branch is a remote object whose interface, *Branch*, provides create, lookup, branchTotal operations.

Introduction

Simple synchronization (without transactions) Atomic operations at the server

- Client operations can interfere.
- Synchronization Without Transactions
  - Object methods should be designed accordingly in a multi-threaded context.
  - If they are not designed in multi-threaded context, strange interleavings can happen.

Introduction

Simple synchronization (without transactions) Atomic operations at the server

- Java's "synchronized" keyword controls threads' access to an object (one thread at a time)
  - When a thread calls a synchronized method, the object is locked, blocking other threads until lock release.
  - Without synchronization, if concurrent *deposit* operations read a balance, then the incremented value will reflect a single deposit.
- Atomic Operations - No thread interference.
  - Built using synchronized methods

Introduction

Enhancing client cooperation by synchronization of server operations

- Clients use a server for resource sharing.
  - some clients update server objects, others access them.
- Synchronized access enables safe resource sharing.
- But communication between threads may be necessary for completion.
- Achieved with *wait* and *notify*.
  - Wait Method - Suspends own execution to allow others to execute.
  - Notify Method - Alerts waiting threads.
- Atomic access even during waits.

Introduction

Enhancing client cooperation by synchronization of server operations

- Consider shared object *Queue*
  - *first* method - Removes first object.
  - *append* method - Adds to queue.
  - Call *first* on empty queue - Wait for addition.
  - *append* calls *notify* - Alerts waiting threads.
  - If threads sync with wait and notify, server answers requests that cannot immediately be satisfied.
- Without synchronization, client need to poll (inefficient and unfair).

Introduction

Failure model for transactions

- Claims of the model:
  - Predictable faults can be handled.
  - Errors can be detected and managed before issues arise.
- Permanent Storage Failures:
  - Writes may fail - Incorrect or missing data.
  - Reads can detect bad data (e.g. by checksums).
- Server Crashes:
  - New process replaces it (no recovery from any volatile memory).
  - Recovery uses permanent storage and info from other processes.
  - Crashes happen also during recovery
- Communication Issues:
  - Messages may suffer delays, duplication, loss, or corruption.
  - Checksums can detect corruption.

Transactions

Transaction T:
a.withdraw(100);
b.deposit(100);
c.withdraw(200);
b.deposit(200);

- Client operations on accounts A, B, C
- Transfer via withdrawal followed by deposit

- \$100 transferred from A to B
- \$200 transferred from C to B

Transactions

All or nothing, isolation

- Transactions apply to recoverable objects and are atomic.
- Atomicity: All or Nothing, i.e. complete or no effect
  - Failure Atomicity: Effects are atomic
  - Durability: successful transactions are stored in permanent memory
- Isolation: No interference between parallel transactions, intermediate effects hidden

Transactions

- To satisfy all-or-nothing and isolation requirements,
  - Objects must be recoverable
  - Changes must be stored in permanent storage
  - Operations must be synchronized
- Serial execution of transactions
  - Unacceptable in most cases

Transactions

Acid properties

- ACID properties mnemonic
  - Atomicity: All or nothing
  - Consistency: Transition between two consistent states
  - Isolation: No mutual effects between transactions
  - Durability: Persist post-commit

Transactions

Operations in the *Coordinator* interface

openTransaction() $\rightarrow$ *trans;*
Starts a new transaction and delivers a unique TID *trans*. This identifier will be used in the other operations in the transaction.

closeTransaction(trans) $\rightarrow$ *(commit, abort);*
Ends a transaction: a *commit* return value indicates that the transaction has committed; an *abort* return value indicates that it has aborted.

abortTransaction(trans);
Aborts the transaction.

- Recoverable object server with transaction capabilities
- Coordinator manages transactions
  - Start transaction with *openTransaction* and receive Transaction Identifier (TID)
  - End transaction with *closeTransaction*
  - Abort with *abortTransaction*
- Transactions need cooperation among client, objects, coordinator
- Clients send TID with each operation on recoverable objects.

Transactions

Successful	*Aborted by client*		*Aborted by server*
openTransaction	*openTransaction*		*openTransaction*
operation	*operation*		*operation*
operation	*operation*		*operation*
•	•	server aborts	•
•	•	*transaction* →	•
operation	*operation*		*operation ERROR*
			reported to client
closeTransaction	*abortTransaction*		

- A transaction is either successful or aborted.
  - Successful: *closeTransaction* by client, changes become permanent, reply is *committed*.
  - Aborted: no effects visible.
- Two ways to abort:
  - Client aborts the transaction calling *abortTransaction*.
  - Server aborts the transaction.

Transactions

Client actions related to server process crashes

- Server actions against crashes
  - if server crashes, replace server process, abort uncommitted transactions, restore objects.
  - if client crashes, give expiry times to transactions.
- Client actions against crashes
  - if server crashes, operation returns an exception after a timeout.
  - Even if the server process is replaced, the old transaction is not valid.

Transactions

Concurrency control - The lost update problem

- Lost Update Problem.
- Concurrent updates lead to loss of information.
- Bank accounts A (\$100), B (\$200), C (\$300).
  - Transaction T: A to B.
  - Transaction U: C to B.
  - Amount is 10% of B.
- Expected balance: \$242 ($200 \rightarrow 220 \rightarrow 242$).
- With concurrent execution, balance becomes \$220.

Transactions

Inconsistent retrievals

Transaction *T*:		Transaction *U*:	
balance = b.getBalance(); *b.setBalance(balance*1.1);* *a.withdraw(balance/10)*		*balance = b.getBalance();* *b.setBalance(balance*1.1);* *c.withdraw(balance/10)*	
balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

- Balance of B: $200
- T and U increase B's balance by $20 and $22
- U's update lost

Transactions

Inconsistent retrievals

Transaction *V*:		Transaction *W*:	
a.withdraw(100) *b.deposit(100)*		*aBranch.branchTotal()*	
a.withdraw(100);	\$100		
		total = a.getBalance()	\$100
		total = total + b.getBalance()	\$300
		total = total + c.getBalance()	
b.deposit(100)	\$300	•	
		•	

- Inconsistent Retrievals Problem:
  - Transaction V: transfers from A to B \$100
  - Transaction W: sum of balances of all account in the branch
  - Accounts A and B: \$200
  - BranchTotal: \$300, wrong

Transactions

Inconsistent retrievals - Serial Equivalence

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance()		*balance = b.getBalance()*	
*b.setBalance(balance*1.1)*		*b.setBalance(balance*1.1)*	
a.withdraw(balance/10)		*c.withdraw(balance/10)*	
balance = b.getBalance()	$200		
*b.setBalance(balance*1.1)*	$220		
		balance = b.getBalance()	$220
		*b.setBalance(balance*1.1)*	$242
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$278

- Serial Equivalence:
  - If two transactions are correct individually, their sequential execution will also be correct.
  - Interleaving operations can be serially equivalent if the overall effect remains the same.
- Avoid lost updates
- Later transaction reads updated

Transactions

Inconsistent retrievals - Serial Equivalence

Transaction *V*:		**Transaction *W*:**	
a.withdraw(100); *b.deposit(100)*		*aBranch.branchTotal( )*	
a.withdraw(100);	$100		
b.deposit(100)	$300		
		total = a.getBalance( )	$100
		total = total + b.getBalance()	$400
		total = total + c.getBalance()	
		...	

- Impact on inconsistent retrievals:
  - Fund transfer in V and branch total in W
  - If retrieval runs along update, then inconsistent
  - If retrieval is before/after update, then consistent

Transactions

Conflicting operations

Operations of different transactions		*Conflict*	*Reason*
read	*read*	No	Because the effect of a pair of *read* operations does not depend on the order in which they are executed
read	*write*	Yes	Because the effect of a *read* and a *write* operation depends on the order of their execution
write	*write*	Yes	Because the effect of a pair of *write* operations depends on the order of their execution

- Conflicting operations: the result depends on execution order
- Consider *read* and *write*
  - The effect includes value returned by *read* and value set by *write*.
  - *read* – *write* and *write* – *write* conflict
- Order of conflicting operations:
  - Serial equivalence of two transactions: conflicting operations are executed in the same order for both transactions.

Transactions

Conflicting operations

Transaction *T*:	Transaction *U*:
$x = read(i)$	
$write(i, 10)$	
	$y = read(j)$
	$write(j, 30)$
$write(j, 20)$	
	$z = read(i)$

- Transactions T and U:
  - $T : x = read(i); write(i, 10); write(j, 20);$
  - $U : y = read(j); write(j, 30); z = read(i);$
- Consider the interleaving in the figure.
- Not serially equivalent.
  - Swapped writes change value.
  - Fact remains unchanged.
- Serial Equivalence Conditions:
  - T accesses both i, j before U.
  - U accesses both i, j before T.

Transactions

Conflicting operations

- Concurrency Control Protocols: three approaches
  - Locking:
    - Locks are set on objects before access.
    - Only locked transactions can access the object.
    - May cause deadlocks.
  - Optimistic Concurrency Control
    - Free operation until commit.
    - Check conflicts before commit.
    - Abort conflicting transactions.
  - Timestamp Ordering
    - Transactions are timestamped.
    - Operations are compared with timestamps to decide execution order.
    - Delayed operations wait; rejected transactions are aborted.

Transactions

Recoverability from aborts

- Recoverability Requirement: log committed transactions.
  - Aborted transactions: no effect.
- Issues from aborts: dirty reads, premature writes.

Transactions

Recoverability from aborts - Dirty Reads

Transaction *T*:		Transaction *U*:	
a.getBalance()		*a.getBalance()*	
a.setBalance(balance + 10)		*a.setBalance(balance + 20)*	
balance = a.getBalance()	\$100		
a.setBalance(balance + 10)	\$110		
		balance = a.getBalance()	\$110
		a.setBalance(balance + 20)	\$130
		commit transaction	
abort transaction			

- Isolation property: uncommitted states should be invisible.
  - Dirty Reads: reading uncommitted changes.
- Example Scenario
  - T updates account A.
  - U reads account A.
  - T aborts after U commits (dirty read).
- U's commit irreversible.

Transactions

Recoverability of transactions - Cascading aborts

- Recoverability
  - U delays commit until observed transactions (e.g. T) commit.
  - U aborts, if observed transaction aborts.
- Cascading Aborts:
  - If T aborts, U must abort too.
  - Other transactions seeing U must also abort.
  - To avoid cascading aborts, read from committed transactions only.
  - Read operations delayed until transactions commit/abort.

Transactions

Premature writes

Operations of the *Account* interface

deposit(amount)
deposit *amount* in the account

withdraw(amount)
withdraw *amount* from the account

getBalance() $\rightarrow$ *amount*
return the balance of the account

setBalance(amount)
set the balance of the account to *amount*

Operations of the *Branch* interface

create(name) $\rightarrow$ *account*
create a new account with a given name

lookUp(name) $\rightarrow$ *account*
return a reference to the account with the given name

branchTotal() $\rightarrow$ *amount*
return the total of all the balances at the branch

- Premature Writes: Aborts impact interaction of writes.
- T and U modify balance.

- If U aborts, T commits:
  - Balance should be \$105
  - Before image of aborting process (U) is restored
  - U's before image \$105. OK
- If T aborts after U commits:
  - Balance should be \$110

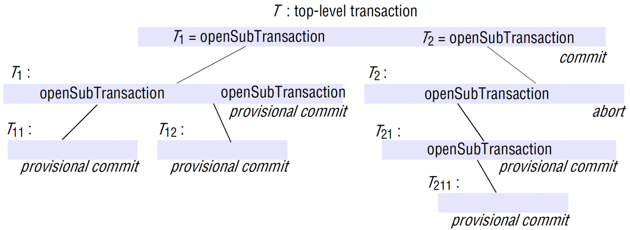
Transactions

Strict executions of transactions - Tentative versions

- Strict Executions:
  - Delay read/write operations
  - Prevents dirty reads and premature writes
  - Waits for commits/aborts
- Tentative Versions:
  - Removable updates
  - Volatile memory storage
  - Transfer upon commit

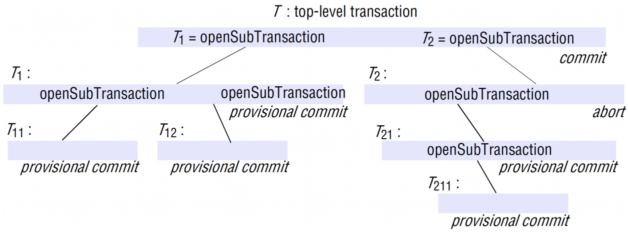
Transactions

Nested transactions until when



- Nested transactions modularity
- Top-level transaction
- Subtransactions:
  - T starts $T1$, $T2$;
  - $T1$ starts $T11$, $T22$;
  - $T2$ starts $T21$, $T211$.

Nested transactions



- Subtransactions ate treated as atomic by parents
- Subtransactions can run concurrently.
- Subtransactions can fail independently.

Nested transactions

- Flat transactions
- Nested Transactions Advantages:
  - Concurrency benefits
  - Independent commit/abort

Nested transactions

- Committing Rules for Nested Transactions:
  - A transaction can only commit or abort after its child transactions are completed.
  - When a subtransaction completes, it makes an independent decision either to commit provisionally or to abort.
  - If a parent transaction aborts, all its subtransactions must abort.
  - A parent can choose to commit even if a subtransaction aborts.
  - If the top-level transaction commits, all provisionally committed subtransactions can also commit, unless their ancestors have aborted.
- Permanent Effects: Not permanent until top-level commit.

Nested transactions

- Top-level aborts on subtransaction abort.
- *Transfer* transaction requires both commits.

$$a.deposit(100)$$

$$b.withdraw(100)$$

  - Both subtransactions needed.
  - Abort of *withdraw* affects parent.
  - Parent abort undoes all.

Locks

- Transactions require serial scheduling.
- Exclusive locks for serialization.
  - Server locks objects that a client is about to use.
  - Another requesting client waits for unlock.

Locks

Transaction T:		**Transaction U:**	
balance = b.getBalance() *b.setBalance(bal*1.1)* *a.withdraw(bal/10)*		*balance = b.getBalance()* *b.setBalance(bal*1.1)* *c.withdraw(bal/10)*	
Operations	Locks	Operations	Locks
openTransaction			
bal = b.getBalance()	lock B		
*b.setBalance(bal*1.1)*		*openTransaction*	
a.withdraw(bal/10)	lock A	*bal = b.getBalance()*	waits for T's lock on B
closeTransaction	unlock A, B	• • •	
			lock B
		*b.setBalance(bal*1.1)*	
		c.withdraw(bal/10)	lock C
		closeTransaction	unlock B, C

- Locking causes wait.
- Unlocking allows access.
- Serial equivalence required.

Locks

- No new locks after release of a lock.
- Growing phase for locks.
- Shrinking phase for release.
- Two-phase locking concept.

Locks

- Delay transactions until previous transactions abort or commit.
- Strict execution concept.
- Strict Two-Phase Locking: Locks until completion.
  - Locks prevent access.
- This way, locks persist until permanent storage...
  - Server holds many objects.
  - Transactions accessed a few, with rare conflicts.
  - Locks limit access.
  - Concurrency granularity matters.

Locks

- Efficient locking scheme needed.
- Many readers/single writer.
- Read lock before a read; write lock before a write.
- Concurrent read operations do not conflict.
- Multiple transactions can share a read lock.
- "Read lock" = "shared lock".

Locks

For one object		*Lock requested*	
		read	*write*
Lock already set	*none*	OK	OK
	read	OK	wait
	write	wait	wait

- Operation conflict rules:
  - Read prevents write.
  - Write prevents read/write.
- Write lock delayed by read lock.
- Read/write lock delayed by write lock.

Locks

- Inconsistent retrievals: Caused by conflicts between read and write operations without concurrency control.
- Prevention of inconsistent retrievals:
  - Retrieval before/after updates.
  - Read locks delay updates, if retrieval comes first.
  - Retrieval is delayed after update, if it comes second.
- Lost updates.
- Prevention of lost updates:
  - Set read locks and then promote them to write.
- Lock promotion:
  - Read lock cannot promote, if it is shared.
  - Request write lock; wait for other read locks to be released.

Locks

For one object		*Lock requested*	
		read	*write*
Lock already set	*none*	OK	OK
	read	OK	wait
	write	wait	wait

- Exclusivity of locks
  - Write locks exclusive
  - No other locks allowed
- Strict Two-Phase Locking Rules:
  - No client access to un/lock operations
  - Locking via read/write requests
  - Unlocking on commit or abort

Locks

Locking rules for nested transactions

- Sets of nested transactions are isolated.
- Transactions in a set are isolated.
  - No concurrent parent-child execution.
  - Parent retains lock transferring it to child temporarily while it executes.
  - Concurrent sibling executions are possible, if they serialize access.

Locks

Locking rules for nested transactions

- Lock rules for acquisition and release:
  - Acquire a read lock, if there is no active write lock.
  - Acquire a write lock, if there is no other locks.
  - Inherited locks from children.
  - Locks from aborted subtransactions are discarded unless retained by the parent.
- Lock acquisition between subtransactions at the same level is possible, if access is serialized.

Locks
Deadlocks

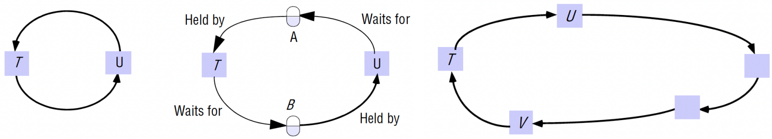
Deadlock with write locks

Transaction T		Transaction U	
Operations	Locks	Operations	Locks
a.deposit(100);	write lock A		
		b.deposit(200)	write lock B
b.withdraw(100)			
•••	waits for U's	*a.withdraw(200);*	waits for T's
	lock on B	•••	lock on A
•••		•••	
•••		•••	

- Deadlock: mutual waiting of two transactions.
- E.g. Deposit and withdraw both acquire write locks.
- T and U are both blocked on locked accounts.

Locks

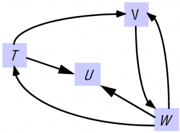
Definition of Deadlock



- Deadlocks can be identified with wait-for graphs
- Nodes are transactions, edges are wait-for relationships.
- Cycles in wait-for graphs indicate deadlocks.
- Breaking a deadlock requires aborting a transaction in the cycle.

Locks

Definition of Deadlock



- Multiple cycles possible:
  - A transactions can be in multiple cycles.
  - Aborting such transactions breaks all involved cycles.

Locks

Deadline prevention - Upgrade locks

- To prevent deadlocks:
  - One can lock all objects used by a transaction initially.
  - This restricts resource access.
  - Predicting necessary locks is difficult.

Locks

Deadlock detection

- Deadlocks: cycles in wait-for graph.
- Abort a transaction to resolve the deadlock.
- Lock Manager's Role in Wait-for Graph Maintenance: adding removing edges based on setLock and unLock operations
- Cycle Checking Frequency:
  - Check cycles every time an edge is added or less frequently.
- Transaction abortion and wait-for graph maintenance:
  - Abort a transaction on deadlock.
  - Remove aborted transaction's node and edges.
  - Choosing the transaction to abort is difficult.

Locks

Timeouts

- Timeouts is one way to handle deadlocks.
- A Lock has a limited time during which it is unbreakable.
  - When timeout expires, lock becomes vulnerable.
  - If there is no competition, the lock remains.
  - With competition, a transaction is aborted.
- Problems with Timeouts:
  - Abortions are possible even without deadlock.
  - Difficulty in determining timeout periods.
- Distributed Deadlocks:
  - Objects across multiple servers complicate deadlock detection.

Locks

Increasing concurrency in locking schemes

- Concurrency in Locking:
  - Increase concurrency despite conflicts.
  - Two approaches:
    - Two-Version Locking: Delay exclusive locks.
    - Hierarchic Locks: Mixed-granularity locks.

Locks

Two version locking

- Optimistic transaction management.
- Higher concurrency potential.
- Types of locks
  - Read Lock: Multiple read locks possible.
  - Write Lock: Single write lock at a time.
  - Commit Lock: Final approval lock.
- Read delayed by commit.

Locks

Two version locking

For one object		*Lock to be set*		
		read	*write*	*commit*
Lock already set	*none*	OK	OK	OK
	read	OK	OK	wait
	write	OK	wait	–
	commit	wait	wait	–

- Rules:
  - Read lock granted unless commit lock exists.
  - Write lock granted unless write/commit lock exists.
  - Commit attempts to convert locks.

Locks

Hierarchic locks

- Suitable for application which require different locking granularities.
  - E.g. some operations require lock at account level or others at branch level.
- Advantages: Mixed granularity reduces overhead and fewer locks needed.
- Disadvantages: The compatibility tables and the rules for promoting locks are more complex.

Optimistic concurrency control

- Disadvantages of Locking:
  - Lock maintenance adds overhead.
  - Locks can lead to deadlocks.
    Cascading aborts.
- Alternative optimistic approach: Low conflict assumption.
- Transactions proceed until a closeTransaction request.
- If a conflict arises at the time of a closeTransaction request, a transaction is aborted and will need to be restarted.

Optimistic concurrency control

- Transaction Phases:
  - Working Phase:
    - Each transaction uses a tentative version of the object.
    - Write operation creates tentative values.
    - Read operation is performed immediately.
  - Validation Phase:
    - Upon a closeTransaction request, the transaction is validated for conflicts.
    - Successful validation allows commitment.
  - Update Phase:
    - Tentative changes become permanent.
    - Read-only transactions commit immediately.
    - Write transactions commit after recording tentative versions in permanent storage.

Optimistic concurrency control

Validation of transactions

- Validation is based on read-write conflict rules
  - Conflicts can occur between *overlapping transactions*.
  - A transaction's overlapping transactions are those that did not commit at the time that transaction started.
- Unique transaction numbering upon closeTransaction request
  - T_i precedes T_j, if $i < j$.

Optimistic concurrency control

Validation of transactions

- Rules for validating a transaction T_v

T_v	T_i	*Rule*	
write	*read*	1.	T_i must not read objects written by T_v.
read	*write*	2.	T_v must not read objects written by T_i.
write	*write*	3.	T_i must not write objects written by T_v and T_v must not write objects written by T_i.

Optimistic concurrency control

Validation of transactions

- Single transaction validation at a time
  - No overlap in update phase
- Validation Types:
  - Backward Validation against preceding overlapping transactions.
  - Forward Validation against succeeding overlapping transactions.

Optimistic concurrency control

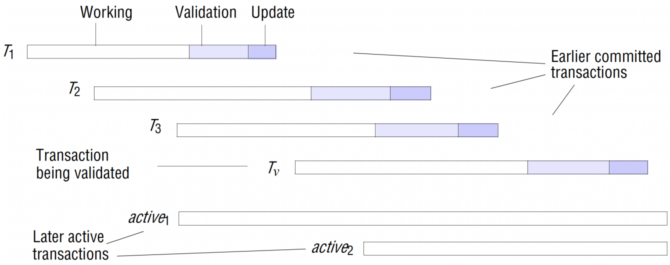
Backward validation

- Backward validation validates transactions after they read data.
- If "startTn" and "finishTn" are largest transaction numbers at the start of working and validation phases, then

boolean valid = true;
for (int T_i = *startTn+1*; T_i <= *finishTn*; T_i++){
    if (read set of T_v intersects write set of T_i) valid = false;
}

Optimistic concurrency control

Backward validation



- T_v to be validated.
- Earlier committed transactions: T_1, T_2, T_3
- T_1 committed before T_v started its working phase.
- Read set of T_v must be compared with the write sets of T_2 and T_3.

Optimistic concurrency control

Backward validation

- Failed validation aborts transaction
- Write sets of old transactions need to be retained until all overlapping transactions validate.

Optimistic concurrency control

Forward validation

- Forward validation compares the write set of T_v with the read sets of overlapping active transactions.

```
boolean valid = true;
for (int T_id = active_1; T_id <= active_N; T_id++){
    if (write set of T_v intersects read set of T_id) valid = false;
}
```

- Compare T_v with active transactions *active*1 to *activeN*.

Optimistic concurrency control

Forward validation

- Conflict Resolution Options:
  - Abort Validating Transaction: Simple, possible unnecessary aborts.
  - Abort Conflicting Transactions: Commit T_v, abort conflicting active transactions.
  - Defer Validation: Wait for conflicts to be resolved.

Optimistic concurrency control

Comparison of forward and backward validation

- Forward vs. backward validation:
  - Forward: flexible conflict resolution.
  - Read sets larger than write sets.
    - Backward: large read sets vs. old write sets.
    - Forward: small write sets vs. active read sets.

Discussion topic

A server manages the objects $a1, a2, \ldots an$. The server provides two operations for its clients:

$read(i)$ returns the value of ai;
$write(i, Value)$ assigns Value to ai.

The transactions T and U are defined as follows:
T: x= read (j); y = read (i); write(j, 44); write(i, 33);
U: x= read(k); write(i, 55); y = read (j); write(k, 66).
Give two serially equivalent interleavings of the transactions T and U.