UNIVERSITY of VIRGINIA

CS4740
CLOUD COMPUTING

Transaction (Contd.)

Prof. Chang Lou, UVA CS, Spring 2024

CONTEXT

- Previously
  - Handling failures: need Atomicity, Durability
  - How? **Write-Ahead Logging**
- Today
  - Handling concurrency: need Isolation
  - How? **Concurrency Control**
    - Pessimistic
    - Optimistic

RECAP: LOST UPDATE PROBLEM

Client 1

```
x = getSeats(ABC123);
  // x = 10
if(x > 1)
  x = x – 1;
write(x, ABC123);
```

Client 2

```
x = getSeats(ABC123);
if(x > 1) //x = 10


  x = x – 1;
write(x, ABC123);
```

At Server: seats = 10

C1's or C2's update was lost!

seats = 9

seats = 9

CONCURRENT TRANSACTIONS

- To prevent transactions from affecting each other
  - A naive solution: could execute them one at a time at server
  - But reduces number of concurrent transactions
  - Transactions per second directly related to revenue of companies
- Goal: increase concurrency while maintaining correctness (ACID)

SERIAL EQUIVALENCE

- An interleaving (say O) of transaction operations is serially equivalent iff (if and only if):
  - There is some ordering (O') of those transactions, one at a time, which
  - Gives the same end-result (for all objects and transactions) as the original interleaving O
  - Where the operations of each transaction occur consecutively (in a batch)
- Says: Cannot distinguish end-result of real operation O from (fake) serial transaction order O'

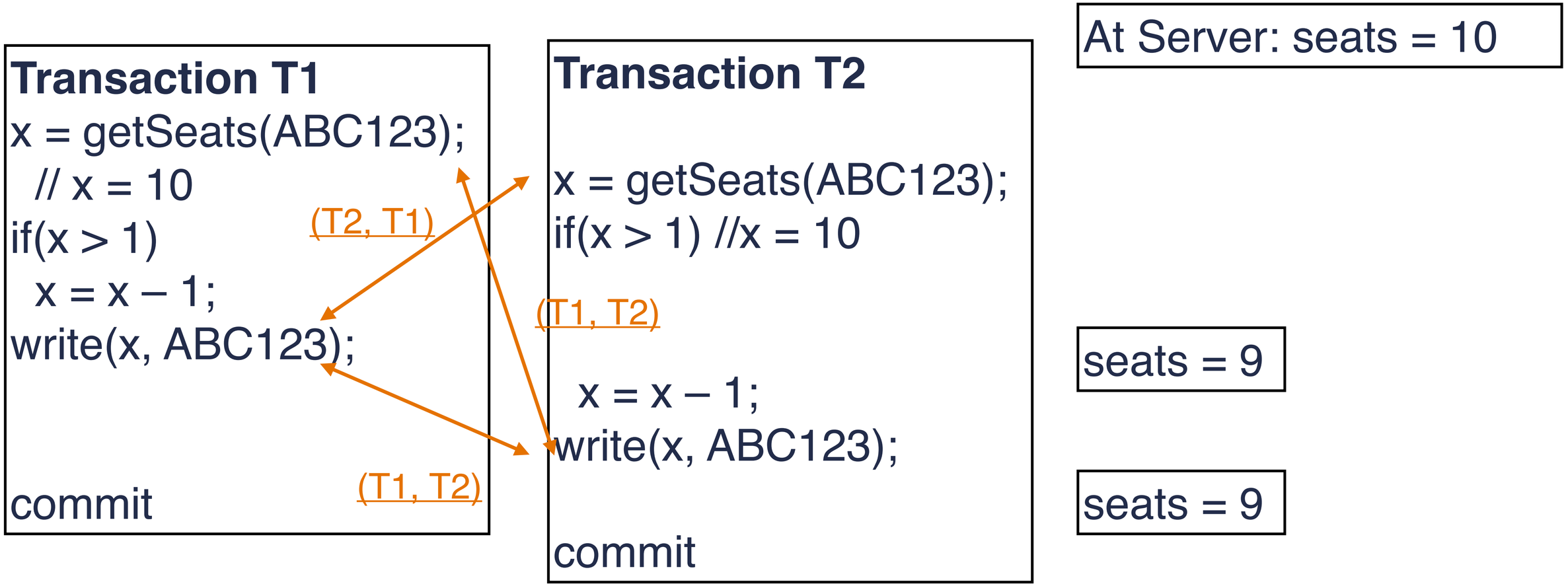
CHECKING FOR SERIAL EQUIVALENCE

- An operation has an effect on
  - The server object if it is a write
  - The client (returned value) if it is a read
- Two operations are said to be conflicting operations, if their combined effect depends on the order they are executed
  - read(x) and write(x)
  - write(x) and write(x)
  - NOT read(x) and read(x): swapping them doesn't change their effects
  - NOT read/write(x) and read/write(y): swapping them ok

CHECKING FOR SERIAL EQUIVALENCE

- Two transactions are serially equivalent if and only if all pairs of conflicting operations (pair containing one operation from each transaction) are executed in the same order (transaction order) for all objects (data) they both access.
  - Take all pairs of conflict operations, one from T1 and one from T2
  - If the T1 operation was reflected first on the server, mark the pair as “(T1, T2)”, otherwise mark it as “(T2, T1)”
  - All pairs should be marked as either “(T1, T2)” or all pairs should be marked as “(T2, T1)”.

1. LOST UPDATE PROBLEM – CAUGHT!



2. INCONSISTENT RETRIEVAL PROBLEM – CAUGHT!

At Server:
ABC123 = 10
ABC789 = 15

Transaction T1

```
x = getSeats(ABC123);
y = getSeats(ABC789);
write(x-5, ABC123);
  // ABC123 = 5 now

write(y+5, ABC789);

commit
```

(T1, T2)

(T2, T1)

Transaction T2

```
x = getSeats(ABC123);
y = getSeats(ABC789);
  // x = 5, y = 15
print("Total:" x+y);
  // Prints "Total: 20"
commit
```

WHAT'S OUR RESPONSE?

- At commit point of a transaction T, check for serial equivalence with all other transactions
  - Can limit to transactions that overlapped in time with T
- If not serially equivalent
  - Abort T
  - Roll back (undo) any writes that T did to server objects

CAN WE DO BETTER?

- Aborting => wasted work
- Can you prevent violations from occurring?

TWO APPROACHES

- Preventing isolation from being violated can be done in two ways
  - **Pessimistic** concurrency control
  - **Optimistic** concurrency control

PESSIMISTIC VS. OPTIMISTIC

- Pessimistic: assume the worst, prevent transactions from accessing the same object
  - E.g., Locking
- Optimistic: assume the best, allow transactions to write, but check later
  - E.g., Check at commit time

BACKGROUND: LOCKING

	Alice	Bob
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	
12:40	Arrive at store.	Look in fridge. Out of milk.
12:45	Buy milk.	Leave for store.
12:50	Arrive home, put milk away.	Arrive at store.
12:55		Buy milk.
1:00		Arrive home, put milk away. Oh no!

BACKGROUND: LOCKING

Alice

```
lock.acquire();
milk++;
lock.release();
```

Bob

```
lock.acquire();
milk++;
lock.release();
```

PESSIMISTIC: EXCLUSIVE LOCKING

- Each object has a lock
- At most one transaction can be inside lock
- Before reading or writing object O, transaction T must call lock(O)
  - Blocks if another transaction already inside lock
- After entering lock T can read and write O multiple times
- When done (or at commit point), T calls unlock(O)
  - If other transactions waiting at lock(O), allows one of them in
- Sound familiar? (This is Mutual Exclusion!)

EXAMPLE

```
TRANSFER(src, dst, x)
01  src_bal = Read(src)
02  if (src_bal > x):
03    src_bal -= x
04    Write(src_bal, src)
05    dst_bal = Read(dst)
06    dst_bal += x
07    Write(dst_bal, dst)

Invocation: TRANSFER(A, B, 50)
```

```
REPORT_SUM(acc1, acc2)
01 acc1_bal = Read(acc1)
02 acc2_bal = Read(acc2)
03 Print(acc1_bal + acc2_bal)


Invocation: PRINT_SUM(A, B)
```

Without transactions: What could go wrong? Think of crashes or inopportune interleavings between concurrent TRANSFER and REPORT_SUM processes.

EXAMPLE

```
TRANSFER(src, dst, x)
00  txID = Begin()
01  src_bal = Read(txID, src)
02  if (src_bal > x):
03    src_bal -= x
04    Write(txID, src_bal, src)
05    dst_bal = Read(txID, dst)
06    dst_bal += x
07    Write(txID, dst_bal, dst)
09    return Commit(txID)
10 Abort(txID)
11 return FALSE
```

```
REPORT_SUM(acc1, acc2)
00  txID = Begin()
01  acc1_bal = Read(txID, acc1)
02  acc2_bal = Read(txID, acc2)
03  Print(acc1_bal + acc2_bal)
04  Commit(txID)
```

LOCK-BASED CONCURRENCY CONTROL

```
TRANSFER(src, dst, x)
00  txID = Begin()
01  src_bal = Read(txID, src)
02  if (src_bal > x):
03    src_bal -= x
04    Write(txID, src_bal, src)
05    dst_bal = Read(txID, dst)
06    dst_bal += x
07    Write(txID, dst_bal, dst)
09    return Commit(txID)
10 Abort(txID)
11 return FALSE
```

```
REPORT_SUM(acc1, acc2)
00  txID = Begin()
01  acc1_bal = Read(txID, acc1)
02  acc2_bal = Read(txID, acc2)
03  Print(acc1_bal + acc2_bal)
04  Commit(txID)
```

Questions: What locks to take, when, and for how long to keep them?

OPTION 1: GLOBAL LOCK FOR ENTIRE TRANSACTION

```
TRANSFER(src, dst, x)
00  txID = Begin()              ← lock(table)
01  src_bal = Read(txID, src)
02  if (src_bal > x):
03    src_bal -= x
04    Write(txID, src_bal, src)
05    dst_bal = Read(txID, dst)
06    dst_bal += x
07    Write(txID, dst_bal, dst)
09    return Commit(txID)       ← unlock(table)
10 Abort(txID)                  ← unlock(table)
11 return FALSE
```

```
REPORT_SUM(acc1, acc2)
00  txID = Begin()              ← lock(table)
01  acc1_bal = Read(txID, acc1)
02  acc2_bal = Read(txID, acc2)
03  Print(acc1_bal + acc2_bal)
04  Commit(txID)                ← unlock(table)
```

Problem?

OPTION 1: GLOBAL LOCK FOR ENTIRE TRANSACTION

```
TRANSFER(src, dst, x)
00  txID = Begin()              ← lock(table)
01  src_bal = Read(txID, src)
02  if (src_bal > x):
03    src_bal -= x
04    Write(txID, src_bal, src)
05    dst_bal = Read(txID, dst)
06    dst_bal += x
07    Write(txID, dst_bal, dst)
09    return Commit(txID)       ← unlock(table)
10 Abort(txID)                  ← unlock(table)
11 return FALSE
```

```
REPORT_SUM(acc1, acc2)
00  txID = Begin()              ← lock(table)
01  acc1_bal = Read(txID, acc1)
02  acc2_bal = Read(txID, acc2)
03  Print(acc1_bal + acc2_bal)
04  Commit(txID)                ← unlock(table)
```

Problem: poor performance.

- Serializes all transactions against that table, even if they don't conflict.

OPTION 2: ROW-LEVEL LOCKS, RELEASE AFTER ACCESS

```
TRANSFER(src, dst, x)
00  txID = Begin()                    ← lock(src)
01  src_bal = Read(txID, src)
02  if (src_bal > x):
03    src_bal -= x
04    Write(txID, src_bal, src)       ← unlock(src)
05    dst_bal = Read(txID, dst)       ← lock(dst)
06    dst_bal += x
07    Write(txID, dst_bal, dst)       ← unlock(dst)
09    return Commit(txID)
10 Abort(txID)
11 return FALSE
```

Problem: insufficient isolation.

- Allows other transactions to read src before dst is updated.

TWO-PHASE LOCKING (2PL)

- Phase 1: acquire locks
  - called "growing phase"
- Phase 2: release locks
  - called "shrink phase"
- You cannot get more locks after you release one.
  - Typically implemented by her releasing locks automatically at end of commit()/ abort().

```
TRANSFER(src, dst, x)
00  txID = Begin()                    ← lock(src)
01  src_bal = Read(txID, src)
02  if (src_bal > x):
03    src_bal -= x
04    Write(txID, src_bal, src)
05    dst_bal = Read(txID, dst)       ← lock(dst)
06    dst_bal += x
07    Write(txID, dst_bal, dst)
09    return Commit(txID)             ← unlock(src,dst)
10 Abort(txID)                        ← unlock(src,dst)
11 return FALSE
```

WHY TWO-PHASE LOCKING => SERIAL EQUIVALENCE?

- Proof by contradiction
- Assume two phase locking system where serial equivalence is violated for some two transactions T1, T2
- Two facts must then be true:
  - (A) For some object O1, there were conflicting operations in T1 and T2 such that the time ordering pair is (T1, T2)
  - (B) For some object O2, the conflicting operation pair is (T2, T1)
  - (A) => T1 released O1's lock and T2 acquired it after that => T1's shrinking phase is before or overlaps with T2's growing phase
- Similarly, (B) => T2's shrinking phase is before or overlaps with T1's growing phase
- But both these cannot be true!

TWO-PHASE LOCKING (2PL)

— Problems?

2PL CAN LEAD TO DEADLOCKS

tx1: lock(foo)	tx2: lock(bar)
tx1: lock(bar)	tx2: lock(foo)

— tx1 might get the lock for foo, then tx2 gets lock for bar, then both transactions wait trying to get the other lock.

PREVENTING DEADLOCK

- Option 1: Each transaction gets all its locks at once.
  - Not always possible (e.g., think foreign key-based navigation in a DB system: rows to lock are determined at runtime).
- Option 2: Each transaction gets its locks in predefined order.
  - As before, not always possible.
- Typically: detect deadlock and abort some transactions as needed to break the deadlock.

DEADLOCK DETECTION AND RESOLUTION

- Construct a waits-for graph:
  - Each vertex in the graph is a transaction.
  - There is an edge T1 → T2 if T1 is waiting for a lock T2 holds.
- There is a deadlock iff there is a cycle in the waits-for graph.
- To resolve, the database unilaterally calls Abort() on one or a few ongoing transactions to break the cycle.
  - How abort() was implemented?
  - **WAL!**

TWO APPROACHES

- Preventing isolation from being violated can be done in two ways
  - **Pessimistic** concurrency control
  - **Optimistic** concurrency control

OPTIMISTIC CONCURRENCY CONTROL

- Increases concurrency more than pessimistic concurrency control
- Increases transactions per second
- For non-transaction systems, increases operations per second and lowers latency
- Used in Dropbox, Google apps, Wikipedia, key-value stores like Cassandra, Riak, and Amazon's Dynamo
- Preferable than pessimistic when conflicts are expected to be rare
  - But still need to ensure conflicts are caught!

FIRST-CUT APPROACH

- Most basic approach
  - Write and read objects at will
  - Check for serial equivalence at commit time
  - If abort, roll back updates made
  - An abort may result in other transactions that read dirty data, also being aborted
    - Any transactions that read from those transactions also now need to be aborted
    - Cascading aborts

ANOTHER APPROACH: TIMESTAMP ORDERING

- Assign unique timestamp to a transaction when it begins
- Each object has two timestamps associated with it:
  - Read timestamp: updated when the object is read
  - Write timestamp: updated when the object is written
- Each transaction has a timestamp = start of transaction
- Good ordering:
  - Object's read and write timestamps will be older than the current transaction if it wants to write an object
  - Object's write timestamps will be older than the current transaction if it wants to read an object
- Abort and restart transaction for improper ordering

BASIC T/O – EXAMPLE 1

SCHEDULE

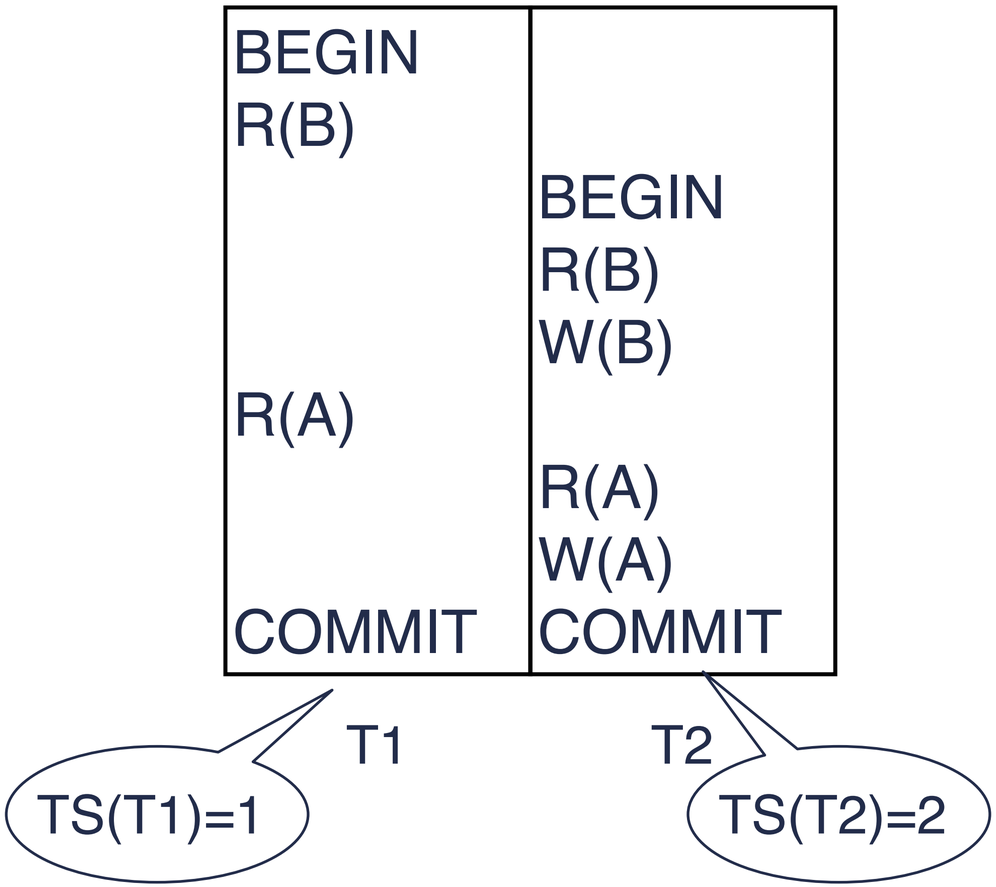
T1	T2
BEGIN	
R(B)	
	BEGIN
	R(B)
	W(B)
R(A)	
	R(A)
	W(A)
COMMIT	COMMIT

DATABASE

OBJECT	R-TS	W-TS
A	0	0
B	0	0

BASIC T/O – EXAMPLE 1

SCHEDULE

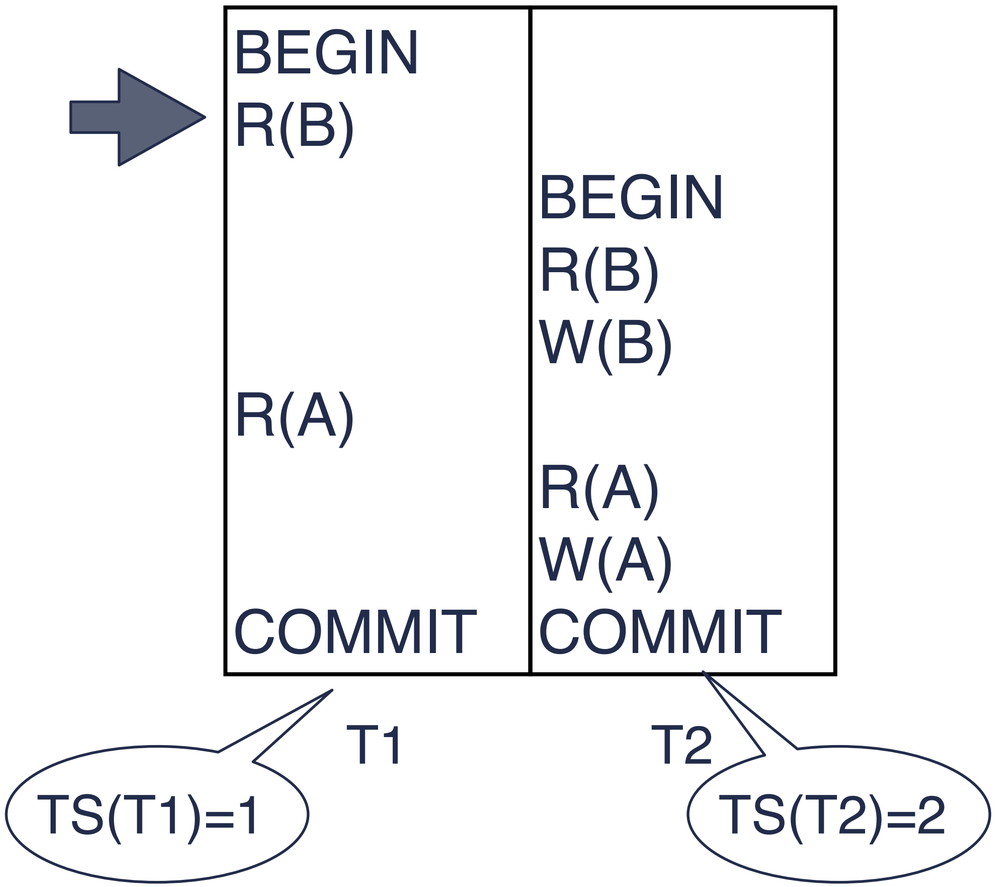


DATABASE

OBJECT	R-TS	W-TS
A	0	0
B	0	0

BASIC T/O – EXAMPLE 1

SCHEDULE

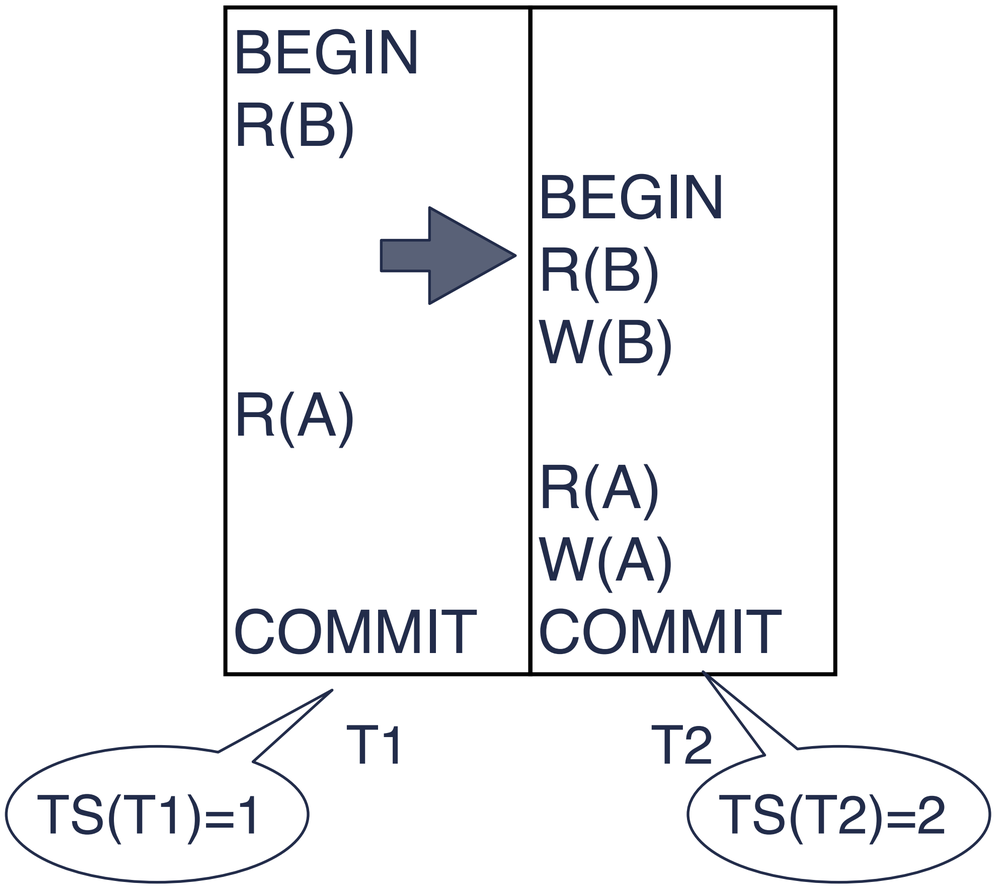


DATABASE

OBJECT	R-TS	W-TS
A	0	0
B	1	0

BASIC T/O – EXAMPLE 1

SCHEDULE

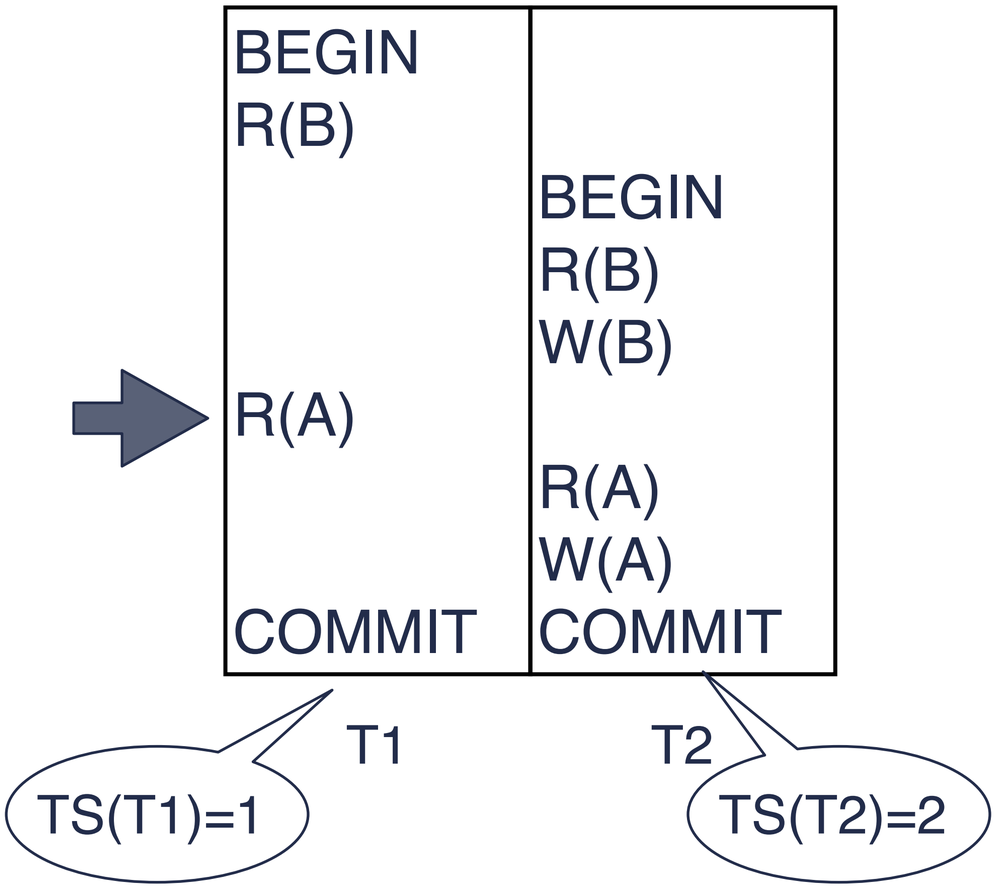


DATABASE

OBJECT	R-TS	W-TS
A	0	0
B	2	0

BASIC T/O – EXAMPLE 1

SCHEDULE

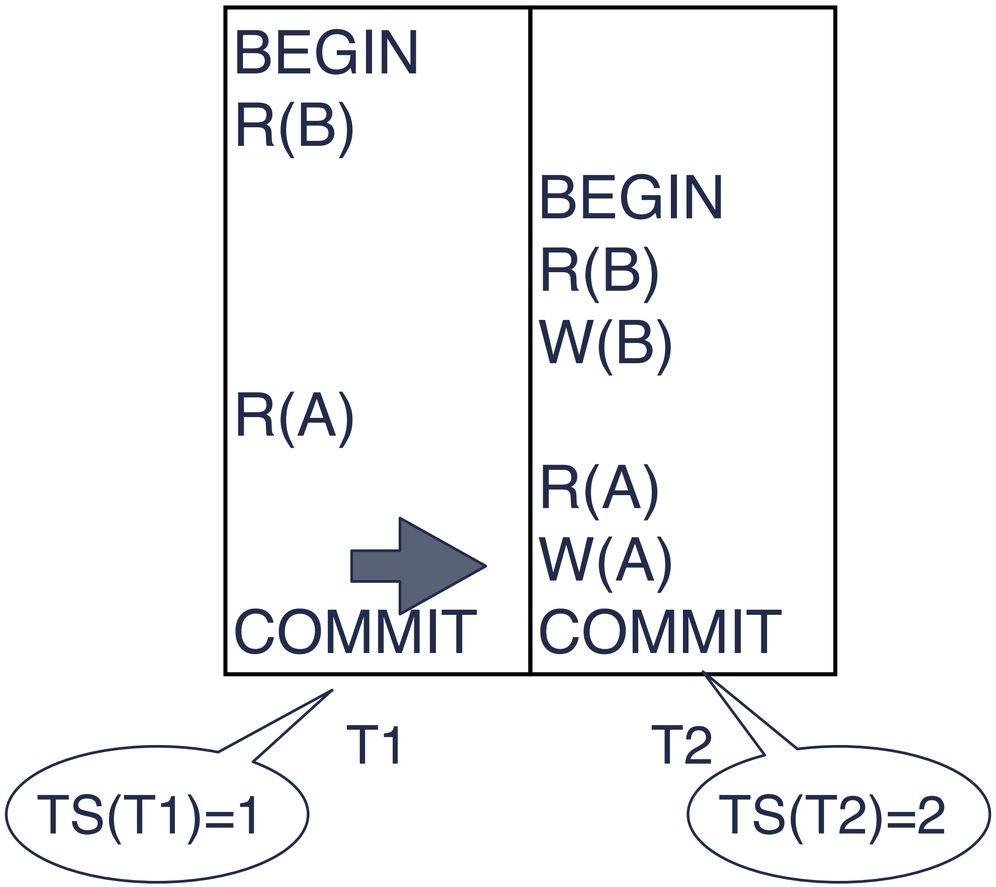


DATABASE

OBJECT	R-TS	W-TS
A	1	0
B	2	2

BASIC T/O – EXAMPLE 1

SCHEDULE



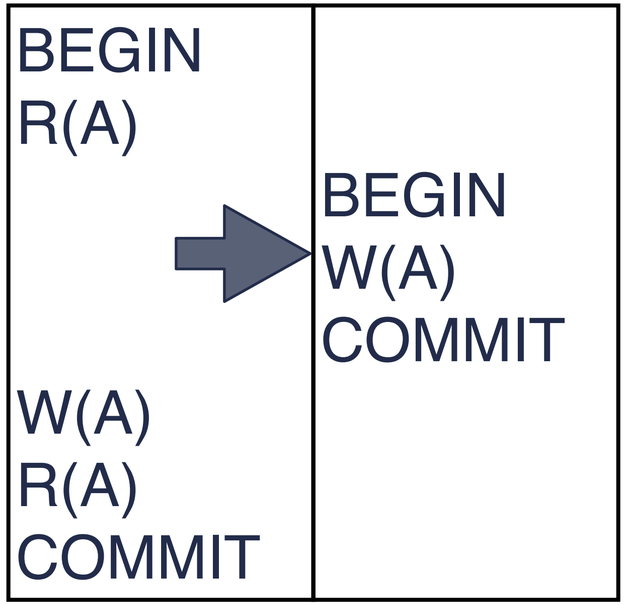
DATABASE

OBJECT	R-TS	W-TS
A	2	2
B	2	2

No violation found! So both txns are safe to commit.

BASIC T/O – EXAMPLE 2

SCHEDULE

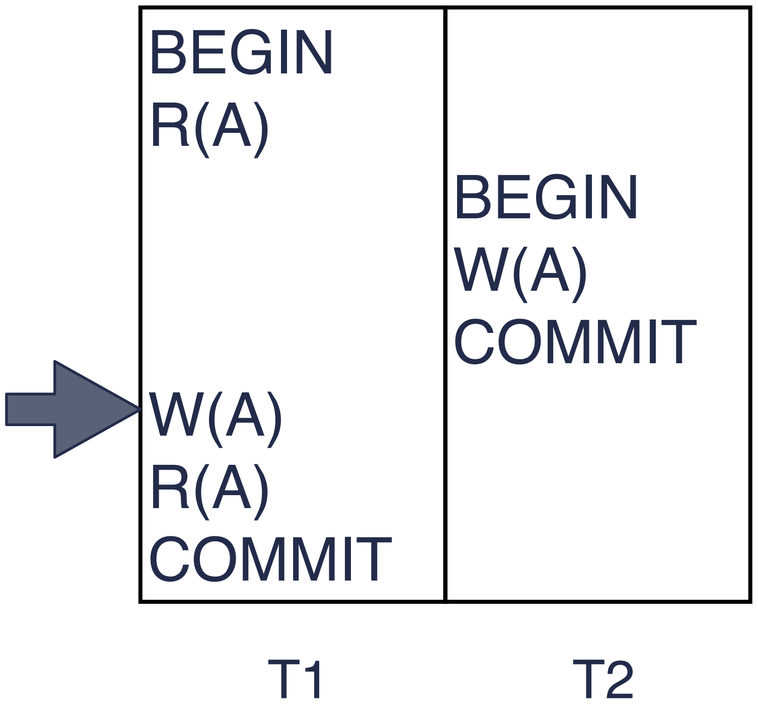


DATABASE

OBJECT	R-TS	W-TS
A	1	2
B	0	0

BASIC T/O – EXAMPLE 2

SCHEDULE



DATABASE

OBJECT	R-TS	W-TS
A	1	2
B	0	0

Violation:
TS(T1)<W-TS(A)

UNIVERSITY of VIRGINIA

TAKEAWAYS

- Transactions need to provide isolation
- Approaches:
  - Pessimistic Concurrency Control
  - Optimistic Concurrency Control
- Next class: **Time and Synchronization**

ACKNOWLEDGEMENT

THIS COURSE IS DEVELOPED HEAVILY BASED ON COURSE MATERIALS SHARED BY PROF. INDRANIL GUPTA, PROF. ROBERT MORRIS, PROF. MICHAEL FREEDMAN, PROF. KYLE JAMIESON, PROF. WYATT LLOYD AND PROF. ROXANA GEAMBASU. MANY APPRECIATIONS FOR GENEROUSLY SHARING THEIR MATERIALS AND TEACHING INSIGHTS.

THIS SLIDES INCLUDES CONTENTS FROM PROF. RYAN HUANG'S OS COURSE (UMICH) AND PROF. JOY ARULRAJ (GATECH)