Concurrency control in transactional systems

Institute of Parallel and Distributed Systems
Rong Chen

*Some slides adapted from Jinyang Li

What we've learnt so far…

- All-or-nothing atomicity in transactions
  - So that failures do not result in (undesirable) intermediate state
- How to realize all-or-nothing atomicity?
  - Logging
  - REDO/UNDO logging vs. REDO-only logging vs. UNDO-only

Concurrency control

- Many application clients are *concurrently* accessing a storage system.
- Concurrent transactions might interfere
  - Similar to data races in parallel programs

An example

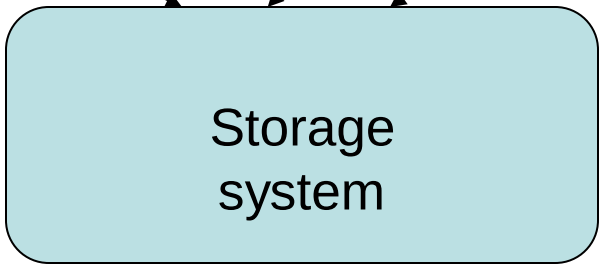
Transfer $100 from A to B

```
A_bal = READ(A)
If (A_bal >100)  {
   B_bal = READ(B)
   B_bal += 100
   A_bal  -= 100
   WRITE(A, A_bal)
   WRITE(B, B_bal)
}
```

Withdraw $50 from A

Report sum of money

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)
   dispense $50 to user
}
```



Solutions?

1. Leave it to application programmers
   - Application programmers are responsible for locking data items
2. Storage system performs automatic concurrency control
   - Concurrent transactions execute as if serially

ACID transactions

- A (Atomicity)
  - All-or-nothing w.r.t. failures
- C (Consistency)
  - Transactions maintain any internal storage state invariants
- I (Isolation)
  - Concurrently executing transactions do not interfere
- D (Durability)
  - Effect of transactions survive failures

Ideal semantics: serializability

- Definition: execution of a set of transactions is equivalent to some serial order
  - Two executions are *equivalent* if they have the same effect on database and produce same output.

Conflict serializability

- An execution schedule is the ordering of read/write/commit/abort operations

```
A_bal = READ(A)
B_bal = READ(B)
B_bal += 100
A_bal  -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)
Commit
```

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)
Commit
```

A (serial) schedule: R(A),R(B),W(A),W(B),C,R(A),R(B),C

Conflict serializability

- Two schedules are equivalent if they:
  - contain same operations
  - order conflicting operations the same way

Two ops conflict if they access the same data item and one is a write.

- A schedule is serializable if it's identical to some serial schedule

Examples

```
A_bal = READ(A)
B_bal = READ(B)
B_bal += 100
A_bal  -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)
```

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)
```

Serializable? R(A),R(B),R(A),R(B),C W(A),W(B),C

Equivalent serial schedule: R(A),R(B),C,R(A),R(B),W(A),W(B),C

Serializable? R(A),R(B), W(A), R(A),R(B),C, W(B),C

Realize a serializable schedule

- Locking-based approach
- Strawman solution 1:
  - Grab global lock before transaction starts
  - Release global lock after transaction commits
- Strawman solution 2:
  - Grab lock on item X before reading/writing X
  - Release lock on X after reading/writing X

Strawman 2

```
A_bal = READ(A)
B_bal = READ(B)
B_bal += 100
A_bal  -= 100
WRITE(A, A_bal)
WRITE(B, B_bal)
```

```
A_bal = READ(A)
B_bal = READ(B)
Print(A_bal+B_bal)
```

Possible with strawman 2? (short-duration locks)

R(A),R(B), W(A), R(A),R(B),C, W(B),C

Locks on writes should be held till end of transaction

Read an uncommitted value

More Strawmans

- Strawman 3
  - Grab write lock on item X before writing X
  - Release write locks at the end of transaction
  - Grab read lock on item X before reading X
  - Release read locks immediately after reading

Read locks must be held till commit time

Non-repeatable reads

R(A), R(A),R(B), W(A), W(B),C, R(B), C

Realize a serializable schedule

- 2 phase locking (2PL)
  - A growing phase in which the transaction is acquiring locks
  - A shrinking phase in which locks are released

2PL: an example

```
R_lock(A)
A_bal = READ(A)
R_lock(B)
B_bal = READ(B)
B_bal += 100
A_bal  -= 100
W_lock(A)
WRITE(A, A_bal)
W_lock(B)
WRITE(B, B_bal)
W_unlock(A)
W_unlock(B)
```

```
R_lock(A)
A_bal = READ(A)
R_lock(B)
B_bal = READ(B)
Print(A_bal+B_bal)
R_unlock(A)
R_unlock(B)
```

Possible? R(A),R(B), W(A), R(A),R(B),C W(B),C

2PL: deadlocks?

- Deadlock avoidance
  - Grab locks in a pre-defined order
  - Not always possible
- Deadlock detection
  - Transaction manager detects deadlock cycles and aborts affected transactions

Disadvantages of locking-based approach

- Need to detect deadlocks
  - Distributed implemention needs distributed deadlock detection (bad)
- Read-only transactions can block update transactions
  - Big performance hit if there are long running read-only transactions

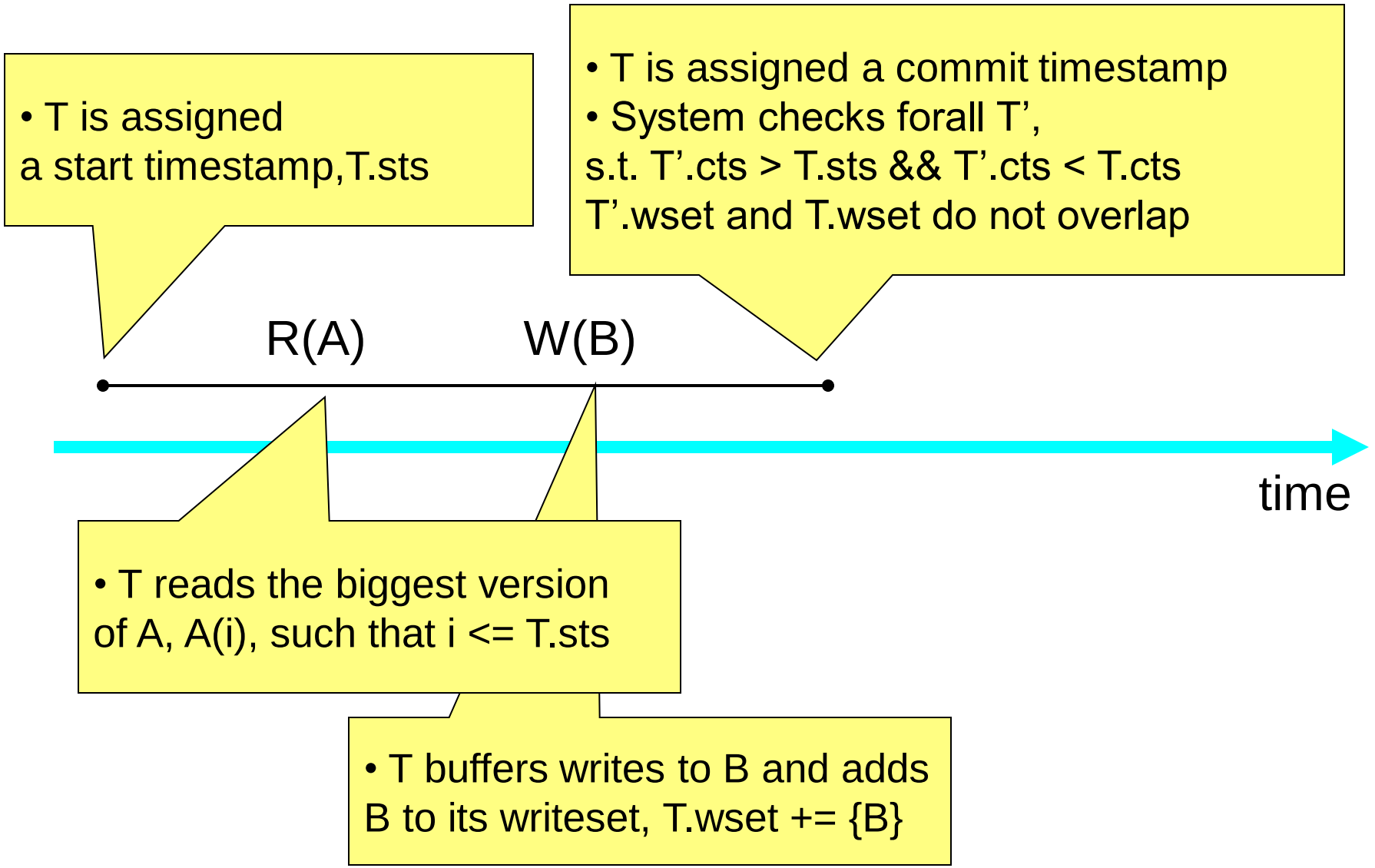
Multi-version concurrency control

- No locking during transaction execution!
- Each data item is associated with multiple versions
- Multi-version transactions:
  - Buffer writes during execution
  - Reads choose the appropriate version
  - At commit time, system validates if okay to make writes visible (by generating new versions)

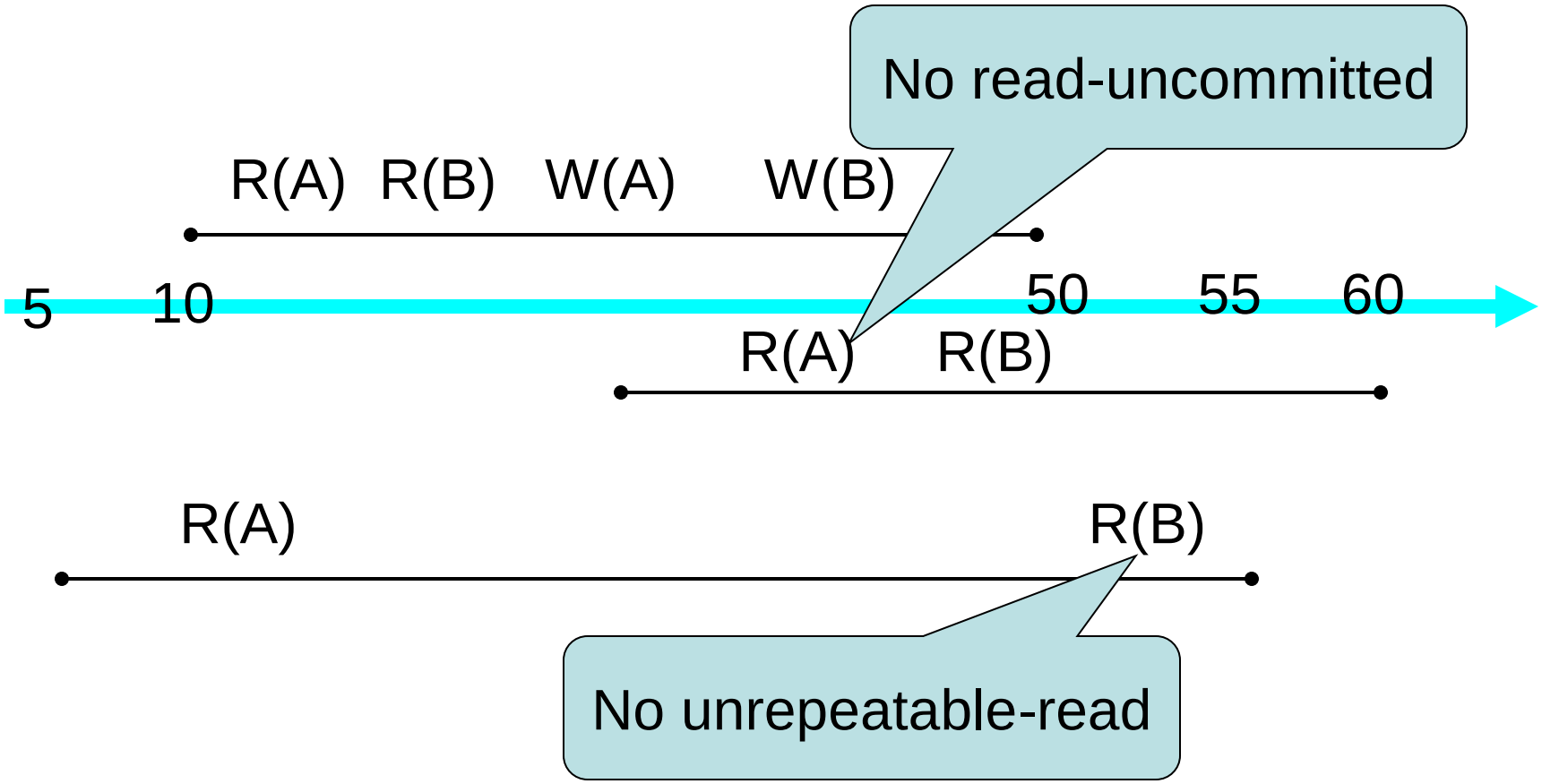
Snapshot isolation

- A popular multi-version concurrency control scheme
- A transaction:
  - reads a “snapshot” of database image
  - Can commit only if there are no write-write conflict

Implementing snapshot isolation



An example


No read-uncommitted
R(A) R(B) W(A) W(B)
5
10
50
55
60
R(A) R(B)
R(A)
R(B)
No unrepeatable-read

Snapshot isolation < serializability

- The write-skew problem

Possible under SI? R(A),R(B),W(B), W(A),C,C

Serializable?

Next Classes

- Fault Tolerance
  - Two phase commit

Thank you !

Any questions?