CS 4440 A

Emerging Database Technologies

Lecture 19

02/12/24

Announcements

- Assignment 1 grade will be released soon
- We will share Assignment 2 feedback next week
- Technology presentation starting from Feb 26
  - Schedule available on the course website
  - Assignments 4,5 due after the last presentation on March 11
  - Attendance required

Using Transactions in SQL

```
SET [GLOBAL | SESSION] TRANSACTION
        transaction_characteristic [, transaction_characteristic] ...

transaction_characteristic: {
        ISOLATION LEVEL level
        | access_mode }

level: {
        REPEATABLE READ
        | READ COMMITTED
        | READ UNCOMMITTED
        | SERIALIZABLE}

access_mode: {
        READ WRITE
        | READ ONLY }
```

Isolation Levels

- With SERIALIZABLE: the interleaved execution of transactions will adhere to our notion of serializability.
- However, if any transaction executes at a lower level, then serializability may be violated.

Dirty reads

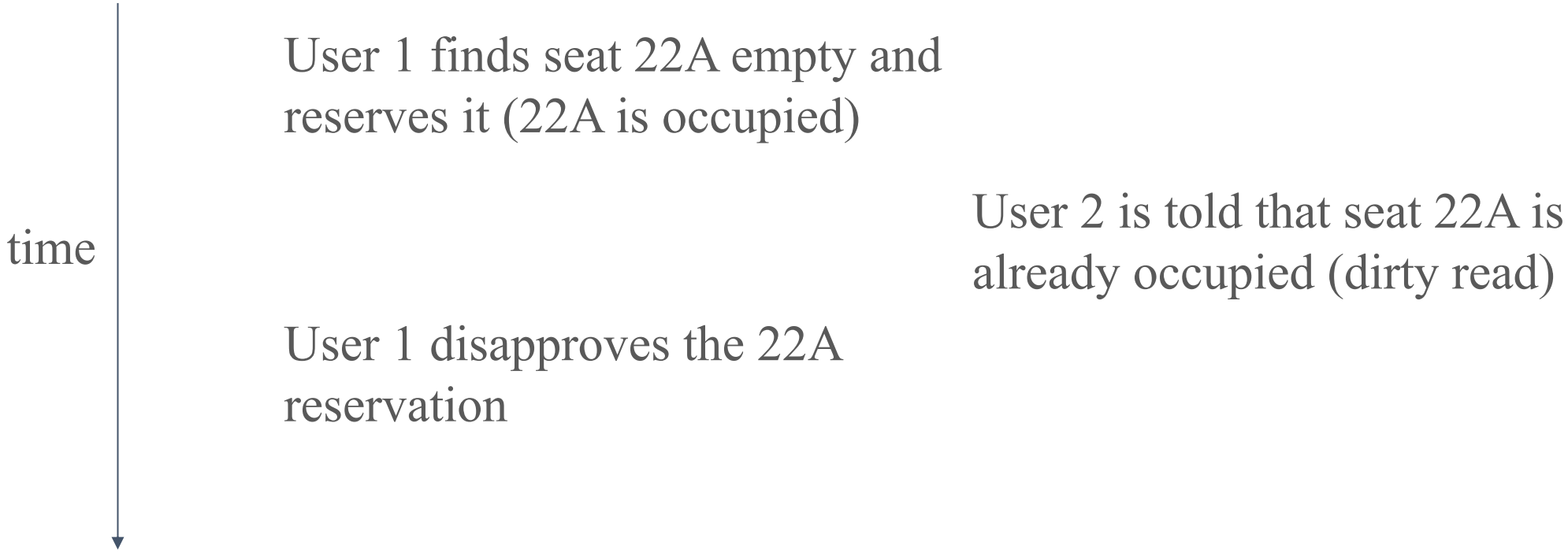
Reading data written by a transaction that has not yet committed

Consider this seat selection example:

1. Find available seat and reserve by setting *seatStatus* to ‘occupied’
2. Ask customer for approval of seat
    a. If so, commit
    b. If not, release seat by setting seatStatus to ‘available’ and repeat Step (1)

Dirty read

- If we allow dirty reads, this can happen



Dirty reads

- If this result is acceptable, the transaction processing can be done faster
  - DBMS does not have to prevent dirty reads
  - Allows more parallelism

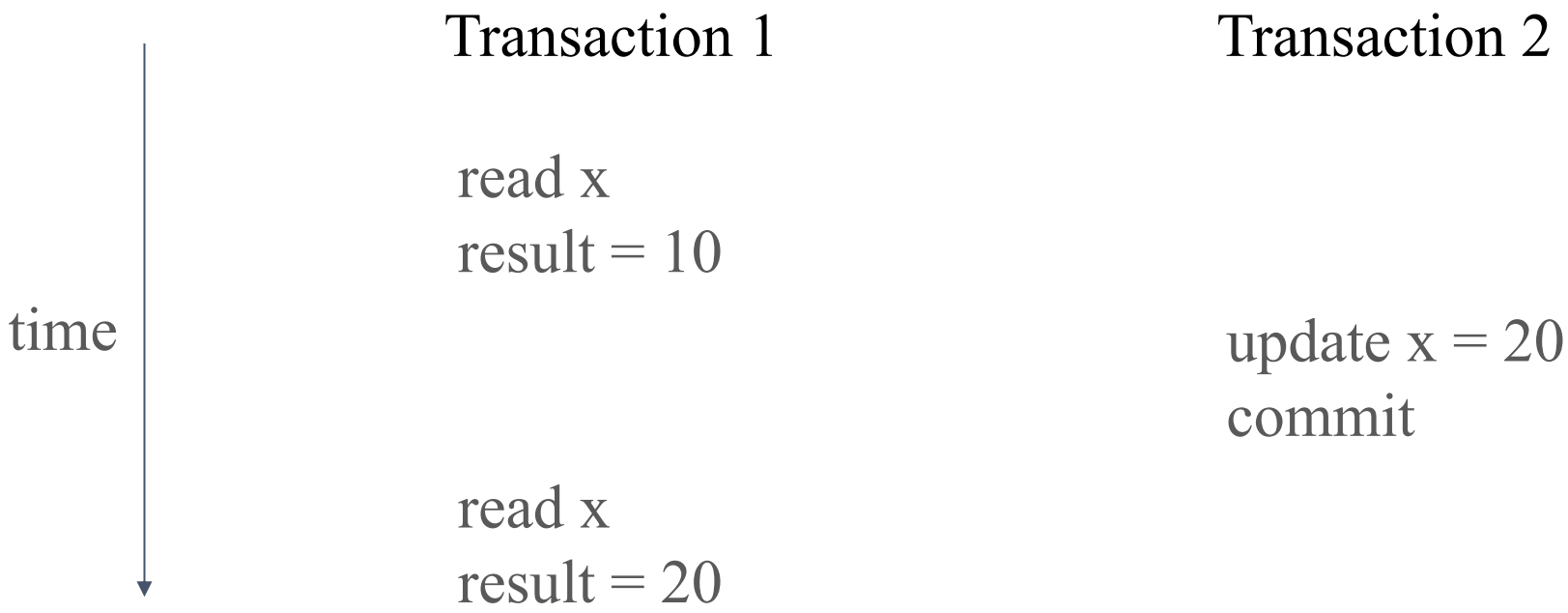
- Tell SQL system:

```
SET TRANSACTION READ WRITE
    ISOLATION LEVEL READ UNCOMMITTED;
```

Read committed

- Only allow reads from committed data, but same query may get different answers

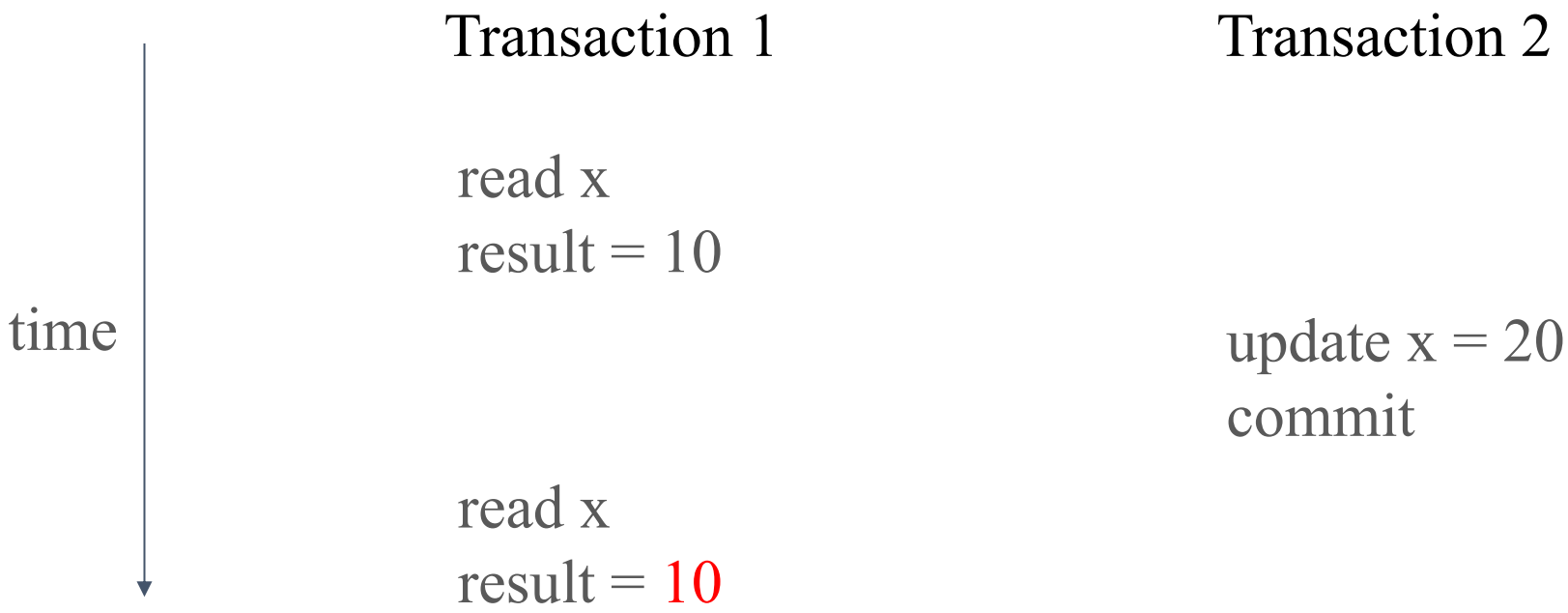
```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```



Repeatable read

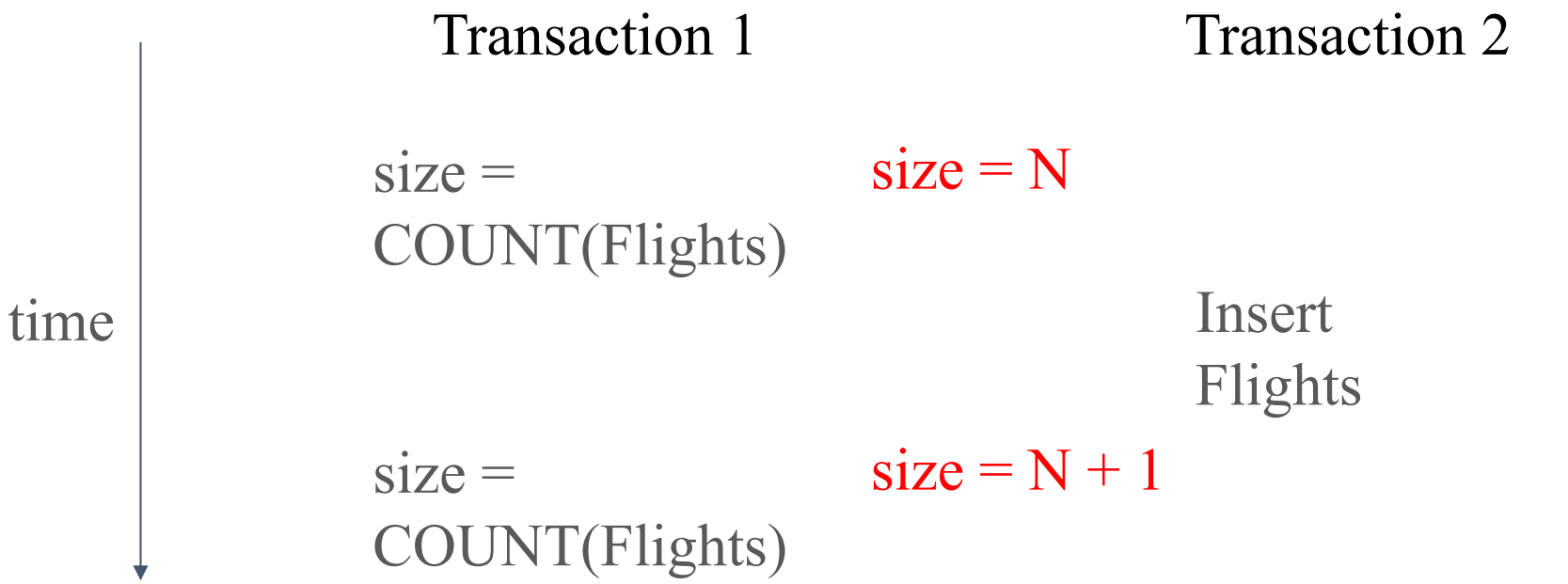
- Any tuple that was retrieved will be retrieved again if the same query is repeated, even though other transactions may modify the individual rows that were read.

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```



Repeatable read

- May allow “phantom” tuples, which are new tuples inserted between queries



Comparison of SQL isolation levels

Isolation Level	Dirty Reads	Nonrepeatable Reads	Phantoms
Read Uncommitted	Allowed	Allowed	Allowed
Read Committed	Not allowed	Allowed	Allowed
Repeatable Read	Not allowed	Not allowed	Allowed
Serializable	Not allowed	Not allowed	Not allowed

Comparison of SQL isolation levels

Isolation Level	Dirty Reads	Nonrepeatable Reads	Phantoms
Read Uncommitted	Allowed	Allowed	Allowed
Read Committed	Not allowed	Allowed	Allowed
Repeatable Read	Not allowed	Not allowed	Allowed
Serializable	Not allowed	Not allowed	Not allowed

- Rarely used in practice, as the performance is not much better than other levels
- In fact, PostgreSQL doesn't support this isolation level
- No lock on data

Comparison of SQL isolation levels

Isolation Level	Dirty Reads	Nonrepeatable Reads	Phantoms
Read Uncommitted	Allowed	Allowed	Allowed
Read Committed	Not allowed	Allowed	Allowed
Repeatable Read	Not allowed	Not allowed	Allowed
Serializable	Not allowed	Not allowed	Not allowed

- Fast and simple to use; adequate for many applications
- Shared lock (read lock) on rows when they are read, exclusive lock (write lock) on rows when they are being modified

Comparison of SQL isolation levels

Isolation Level	Dirty Reads	Nonrepeatable Reads	Phantoms
Read Uncommitted	Allowed	Allowed	Allowed
Read Committed	Not allowed	Allowed	Allowed
Repeatable Read	Not allowed	Not allowed	Allowed
Serializable	Not allowed	Not allowed	Not allowed

- Good for reporting, data warehousing types of workload
- Shared locks on all rows read by a transaction

Comparison of SQL isolation levels

Isolation Level	Dirty Reads	Nonrepeatable Reads	Phantoms
Read Uncommitted	Allowed	Allowed	Allowed
Read Committed	Not allowed	Allowed	Allowed
Repeatable Read	Not allowed	Not allowed	Allowed
Serializable	Not allowed	Not allowed	Not allowed

- Recommended only when updating transactions contain logic sufficiently complex that they might give wrong answers in Read Committed mode
- Locking the entire range of rows that could potentially be accessed by a transaction's queries

Recap: ACID properties

- **Atomicity**: A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
- **Consistency**: A correct execution of the transaction must take the database from one consistent state to another.
- **Isolation**: A transaction should not make its updates visible to other transactions until it is committed.
- **Durability**: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

This class: ensuring isolation via concurrency control

Reading Materials

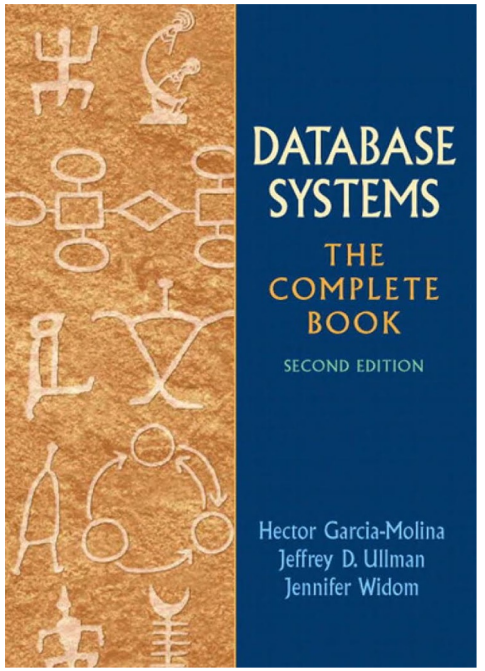
Database Systems: The Complete Book (2nd edition)

- Chapter 18 – Concurrency Control

Supplementary materials

Fundamental of Database Systems (7th Edition)

- Chapter 21 - Concurrency Control Techniques



Acknowledgement: The following slides have been adapted from EE477 (Database and Big Data Systems) taught by Steven Whang.

Schedule

An actual or potential sequence for executing actions as seen by the DBMS

A list of actions from a set of transactions

- includes READ, WRITE, ABORT, COMMIT

Two actions from the same transaction T MUST appear in the schedule in the same order that they appear in T

- cannot reorder actions from a given transaction

Assumptions

Transactions communicate only through READ and WRITE

- i.e., no exchange of message among them

A database is a “fixed” collection of independent objects

- i.e., objects are not added to or deleted from the database
- this assumption could be relaxed

Transaction primitives

INPUT(X) : copy block X from disk to memory

READ(X, *t*): copy X to transaction's local variable *t*
(run INPUT(X) if X is not in memory)

WRITE(X, *t*): copy value of *t* to X (run INPUT(X) if X is not in memory)

OUTPUT(X): copy X from memory to disk

Schedule

- Actions taken by one or more transactions

T1	*T2*
READ(A, t)	READ(A, s)
t := t+100	s := s*2
WRITE(A, t)	WRITE(A, s)
READ(B, t)	READ(B, s)
t := t+100	s := s*2
WRITE(B, t)	WRITE(B, s)

Characterizing Schedules based on Serializability (1)

Serial schedule:

- A schedule S is serial if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule.
  - Otherwise, the schedule is called nonserial schedule.

Serializable schedule:

- A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions.

Serial schedule

- One transaction is executed at a time

T1	*T2*	*A*	*B*
		25	25
READ(A, t)			
$t := t+100$			
WRITE(A, t)		125	
READ(B, t)			
$t := t+100$			
WRITE(B, t)			125
	READ(A, s)		
	$s := s*2$		
	WRITE(A, s)	250	
	READ(B, s)		
	$s := s*2$		
	WRITE(B, s)		250

Schedule: (T1, T2)

Q: Do serial schedules allow for high throughput?

Serializable schedule

- There exists a serial schedule with the same effect

T1	*T2*	*A*	*B*
		25	25
READ(A, t)			
t := t+100			
WRITE(A, t)		125	
	READ(A, s)		
	s := s*2		
	WRITE(A, s)	250	
READ(B, t)			
t := t+100			
WRITE(B, t)			125
	READ(B, s)		
	s := s*2		
	WRITE(B, s)		250

Same effect as (T1, T2)

Serializable schedule

- This is not serializable

T1	*T2*	*A*	*B*
		25	25
READ(A, t)			
t := t+100			
WRITE(A, t)		125	
	READ(A, s)		
	s := s*2		
	WRITE(A, s)	250	
	READ(B, s)		
	s := s*2		
	WRITE(B, s)		50
READ(B, t)			
t := t+100			
WRITE(B, t)			150

Serializable schedule

- Serializable, but only due to the detailed transaction behavior

T1	*T2*	*A*	*B*
		25	25
READ(A, t)			
t := t+100			
WRITE(A, t)		125	
	READ(A, s)		
	s := s+200		
	WRITE(A, s)	325	
	READ(B, s)		
	s := s+200		
	WRITE(B, s)		225
READ(B, t)			
t := t+100			
WRITE(B, t)			325

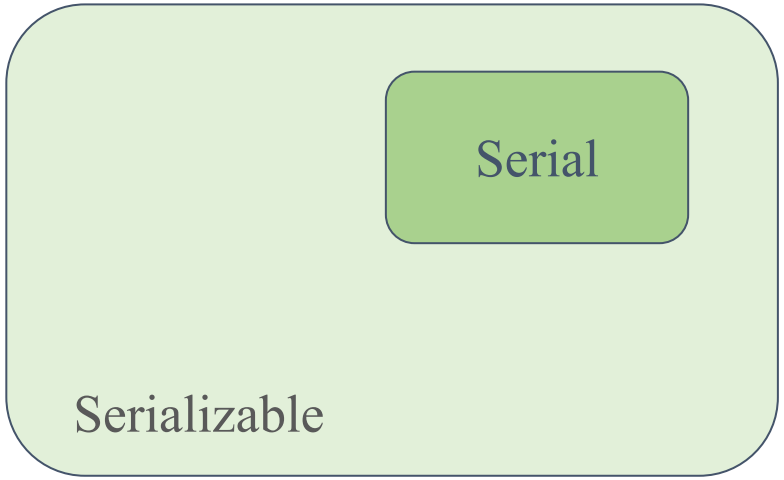
Same effect as (T1, T2)

Serial vs Serializable Schedule

Being serializable is not the same as being serial

Being serializable implies that the schedule is a correct schedule.

- It will leave the database in a consistent state.
- The interleaving is appropriate and will result in a state as if the transactions were serially executed, yet will achieve efficiency due to concurrent execution.



Notation for transactions and schedule

Serializability is hard to check - cannot always know detailed behaviors

Abstract view of transactions:

- $r_i(X)$: Ti reads *X*
- $w_i(X)$: Ti writes *X*

T1: $r_1(A); w_1(A); r_1(B); w_1(B)$

T2: $r_2(A); w_2(A); r_2(B); w_2(B)$

Serializable schedule: $r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B);$

Conflicts

- A pair of consecutive actions that cannot be interchanged without changing behavior

These are conflicts	These are not conflicts
$r_i(X); w_i(Y)$	$r_i(X); r_j(X)$
$r_i(X); w_j(X)$	$r_i(X); w_j(Y)$
$w_i(X); r_j(X)$	$w_i(X); r_j(Y)$
$w_i(X); w_j(X)$	$w_i(X); w_j(Y)$

Characterizing Schedules based on Serializability (2)

Conflict equivalent

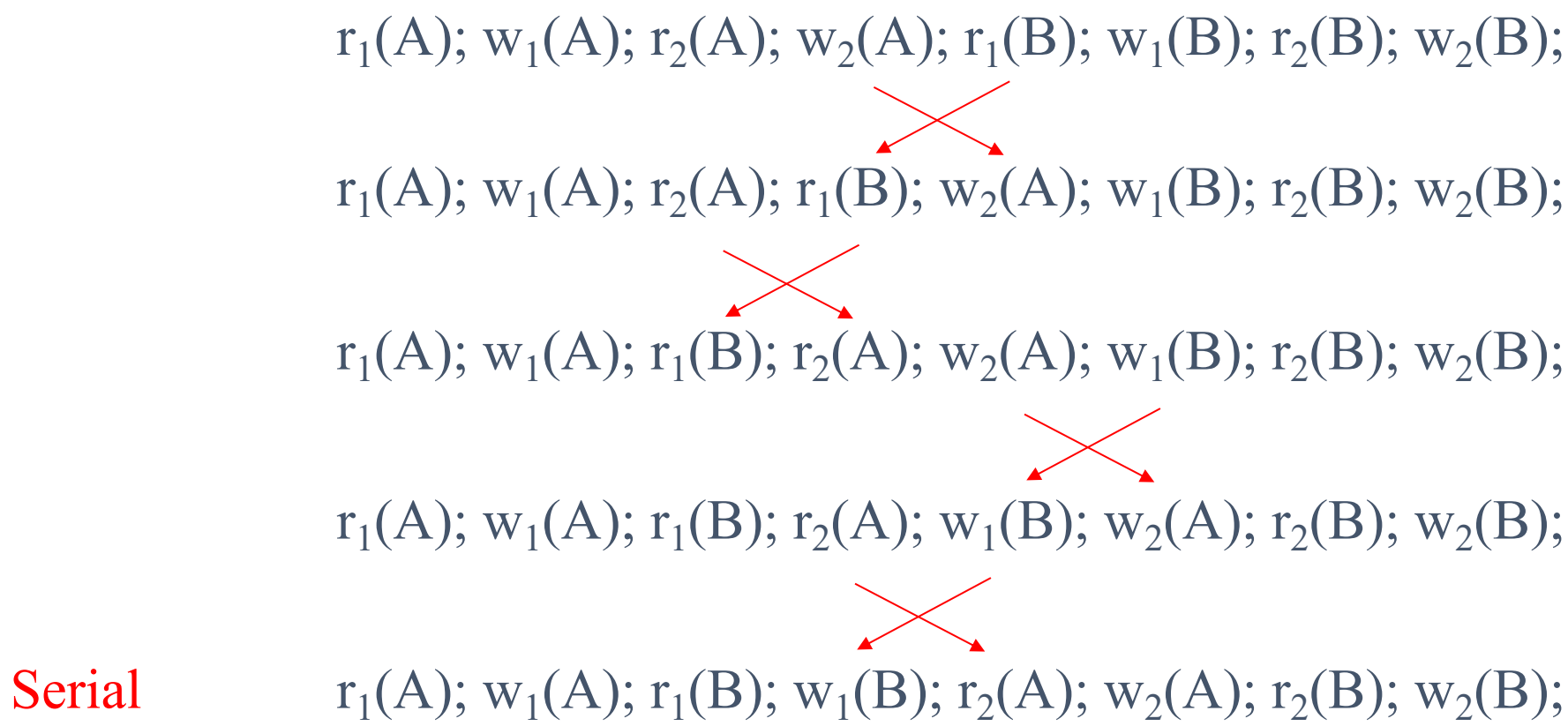
- Two conflict equivalent schedules have the same effect on a database
- All pairs of conflicting actions are in same order
- one schedule can be obtained from the other by swapping “non-conflicting” actions
  - either on two different objects
  - or both are read on the same object

Conflict serializable

- A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S’.

Conflict-serializable schedule

- Conflict-equivalent to serial schedule

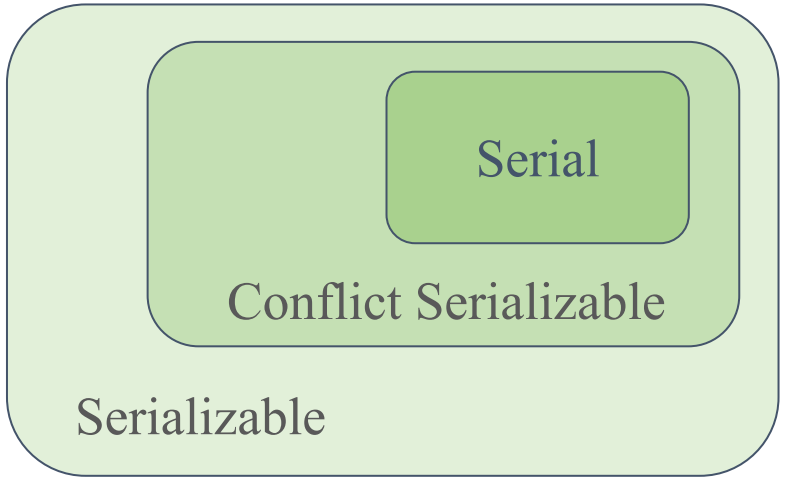


Conflict-serializable schedule

- A conflict-serializable schedule is always serializable
- But not vice versa (e.g., serializable schedule due to detailed transaction behavior)

S1: $w_1(Y); w_1(X); w_2(Y); w_2(X); w_3(X);$ Serial

S2: $w_1(Y); w_2(Y); w_2(X); w_1(X); w_3(X);$ Serializable, but not conflict serializable



Exercise #1

- What are schedules that are conflict-equivalent to (T1, T2)?

T1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$;

T2: $r_2(B)$; $w_2(B)$; $r_2(A)$; $w_2(A)$;

Testing for conflict serializability

Through a *precedence graph* *:

- Looks at only read_Item (X) and write_Item (X) operations
- Constructs a precedence graph - a graph with directed edges
- An edge is created from Ti to Tj if one of the operations in Ti appears before a conflicting operation in Tj
- The schedule is serializable if and only if the precedence graph has no cycles.

* Also called dependency graph, conflict graph, or serializability graph

Precedence graph

Can use to decide conflict serializability

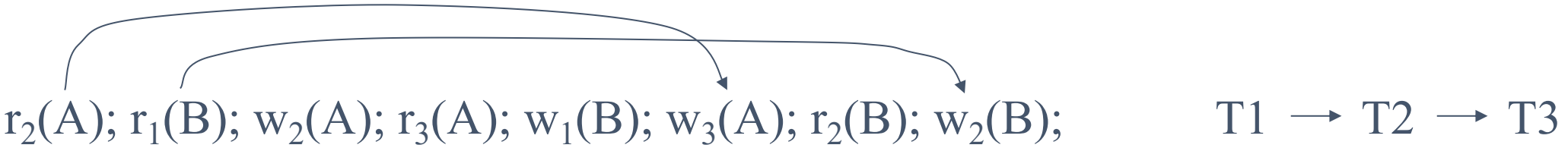
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B);$

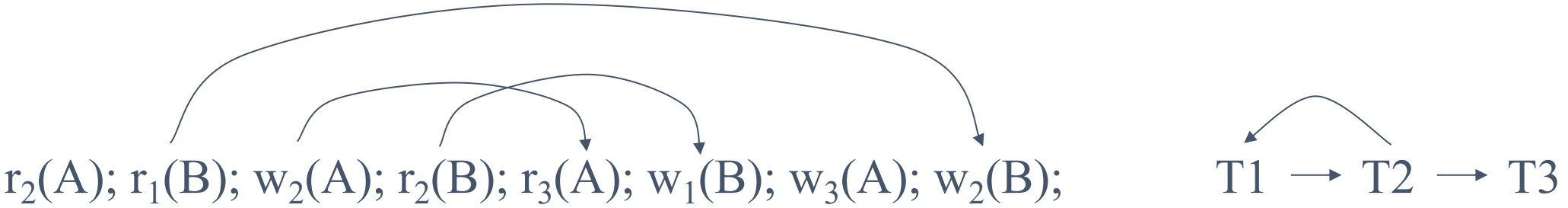
$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B);$

One node per committed transaction
Edge from Ti to Tj if an action of Ti precedes and conflicts with one of Tj's actions
– Wi(A) --- Rj(A), or Ri(A) --- Wj(A), or Wi(A) --- Wj(A)

Precedence graph

Can use to decide conflict serializability





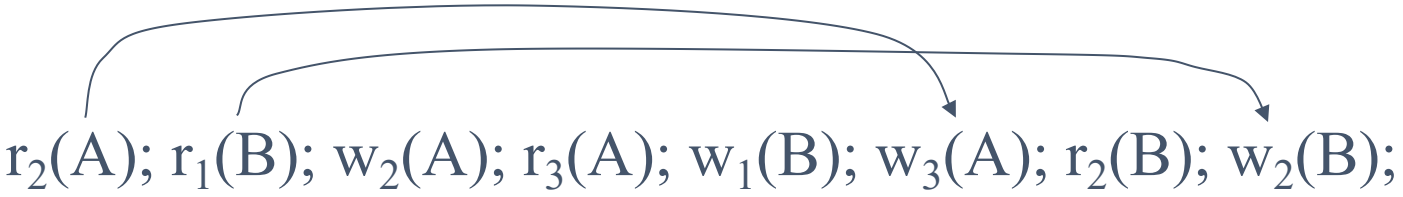
One node per committed transaction

Edge from Ti to Tj if an action of Ti precedes and conflicts with one of Tj's actions

– Wi(A) --- Rj(A), or Ri(A) --- Wj(A), or Wi(A) --- Wj(A)

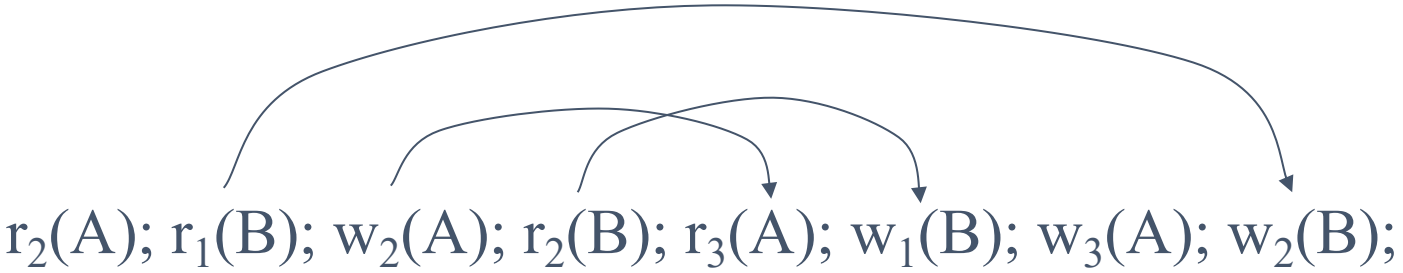
Precedence graph

Can use to decide conflict serializability



This is conflict serializable

T1 → T2 → T3



This is not because of cycle

One node per committed transaction

Edge from Ti to Tj if an action of Ti precedes and conflicts with one of Tj's actions

– Wi(A) --- Rj(A), or Ri(A) --- Wj(A), or Wi(A) --- Wj(A)

Exercise #2

- What is the precedence graph for the schedule:

$$r_1(A); r_2(A); r_1(B); r_2(B); r_3(A); r_4(B); w_1(A); w_2(B);$$

One node per committed transaction
Edge from Ti to Tj if an action of Ti precedes and conflicts with one of Tj's actions
– Wi(A) --- Rj(A), or Ri(A) --- Wj(A), or Wi(A) --- Wj(A)

Schedule Summary

Schedule

- Serial schedule
- Serializable schedule (why do we need them?)
- Conflicting actions
- Conflict-equivalent schedules
- Conflict-serializable schedule

Dependency (or Precedence) graphs

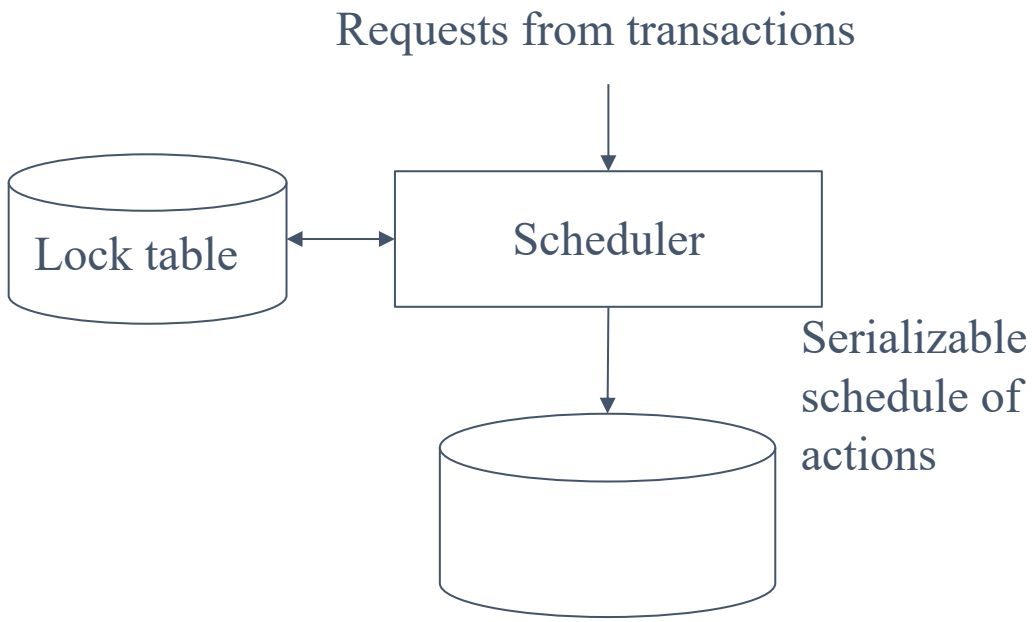
- their relation to conflict serializability (by acyclicity)

Enforce serializability with locks

$l_i(X)$: Ti requests lock on X
$u_i(X)$: Ti releases lock on X

Consistency of transactions
- Can only read/write element if granted a lock
- A locked element must later be unlocked

Legality of schedules
- No two transactions may lock element at the same time



Enforce serializability with locks

- Legal, but not serializable schedule

T1	*T2*	*A*	*B*
		25	25
$l_1(A)$; $r_1(A)$;			
$A := A+100$			
$w_1(A)$; $u_1(A)$;		125	
	$l_2(A)$; $r_2(A)$		
	$A := A*2$		
	$w_2(A)$; $u_2(A)$	250	
	$l_2(B)$; $r_2(B)$		
	$B := B*2$		
	$w_2(B)$; $u_2(B)$		50
$l_1(B)$; $r_1(B)$			
$B := B+100$			
$w_1(B)$; $u_1(B)$;			150

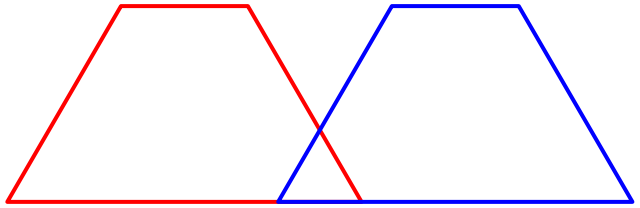
Two-phase locking (2PL)

- In every transaction, all lock actions precede all unlock actions
- Guarantees a legal schedule of consistent transactions is conflict serializable

Two-phase locking (2PL)

- This is now conflict serializable

T1	*T2*	*A*	*B*
		25	25
$l_1(A)$; $r_1(A)$;			
$A := A+100$			
$w_1(A)$; $l_1(B)$; $u_1(A)$;		125	
	$l_2(A)$; $r_2(A)$		
	$A := A*2$		
	$w_2(A)$;	250	
	l_2(B) Denied		
$r_1(B)$; $B := B+100$			
$w_1(B)$; $u_1(B)$;			125
	$l_2(B)$; $u_2(A)$; $r_2(B)$		
	$B := B*2$		
	$w_2(B)$; $u_2(B)$		250



Locking with several modes

Using one type of lock is not efficient when reading and writing

Instead, use shared locks for reading and exclusive locks for writing

$sl_i(X)$: Ti requests shared lock on X
$xl_i(X)$: Ti requests exclusive lock on X

Requirements: analogous notions of consistent transactions, legal schedules, and 2PL

Locking with several modes

- More efficient than previous schedule

T1	*T2*
$sl_1(A)$; $r_1(A)$;	
	$sl_2(A)$; $r_2(A)$;
	$sl_2(B)$; $r_2(B)$;
$xl_1(B)$ **Denied**	
	$u_2(A)$; $u_2(B)$;
$xl_1(B)$; $r_1(B)$; $w_1(B)$;	
$u_1(A)$; $u_1(B)$;	

- T1 and T2 can read *A* at the same time
- T1 and T2 use 2PL, so the schedule is conflict serializable

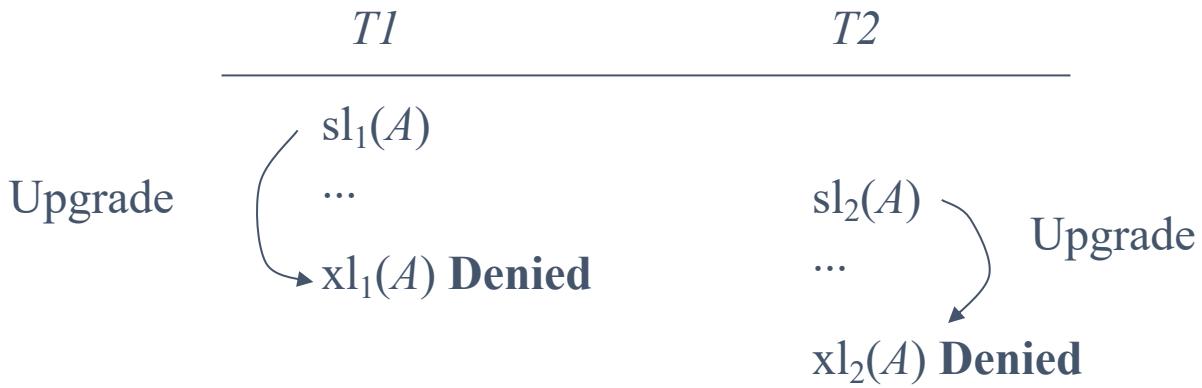
Locking with several modes

- Compatibility matrix

		Lock requested	
		S	X
Lock held in mode	S	Yes	No
	X	No	No

Update locks

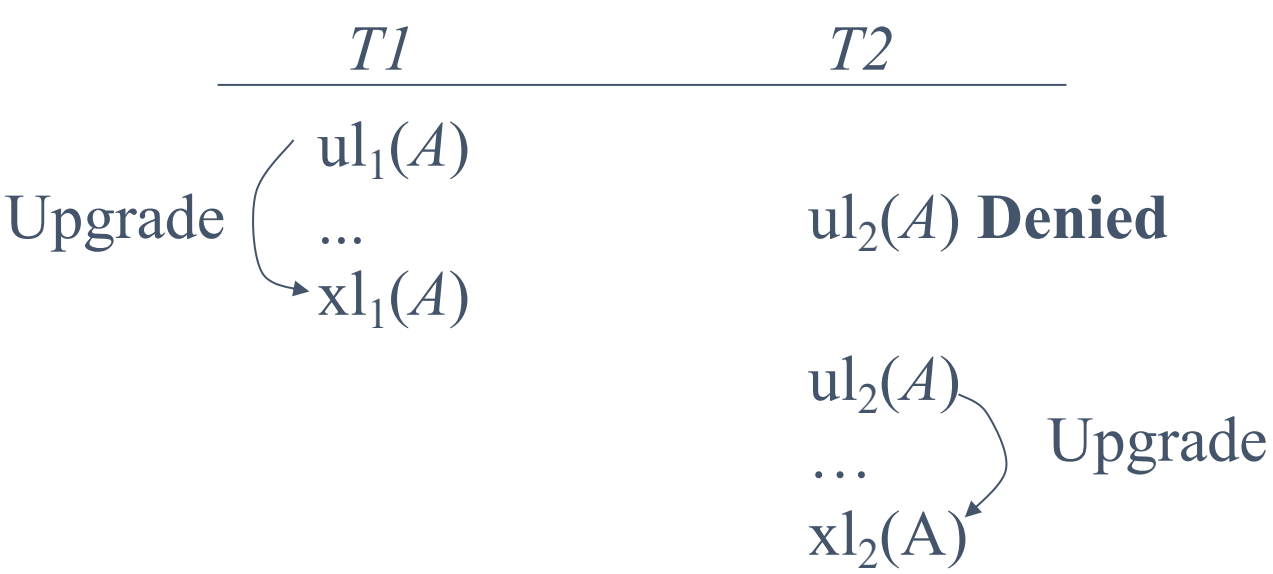
- If T reads and writes the same X, enable lock to upgrade from shared to exclusive
  - Obviously allows more parallelism
- However, a simple upgrading approach may lead to deadlocks



Update locks

$ul_i(X)$: Ti requests an update lock on X

- Solution: introduce new type called update locks
- Only an update lock can be updated to an exclusive lock later

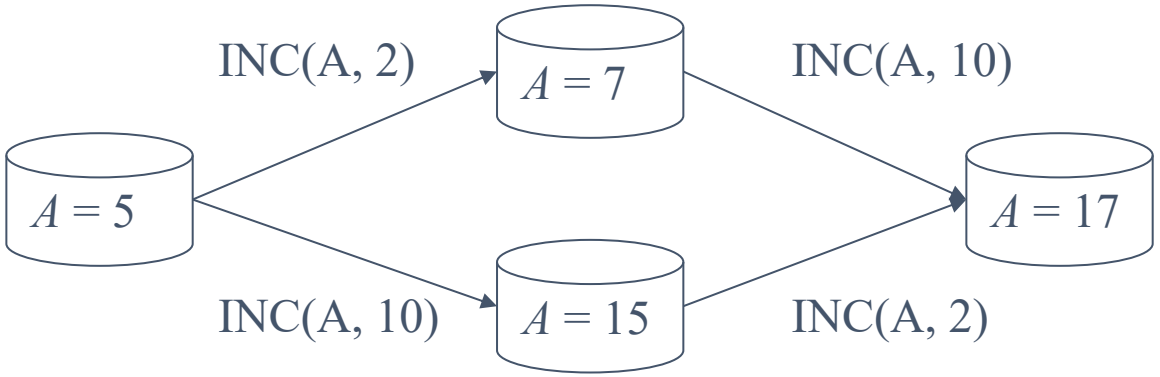


Compatibility matrix

	S	X	U
S	Yes	No	Yes
X	No	No	No
U	No	No	No

Increment locks

- Many transactions only increment or decrement values
  - E.g., bank account transfer
- Introduce increment locks just for this purpose



Compatibility matrix

	S	X	I
S	Yes	No	No
X	No	No	No
I	No	No	Yes