



# Announcements

- HW4 is due on Friday

# Recap: Applications and Databases

Almost every app uses some database

- General purpose language (Java, Python)
- App issues SQL commands to RDBMS
- Usually, multiple apps (users) access same DB

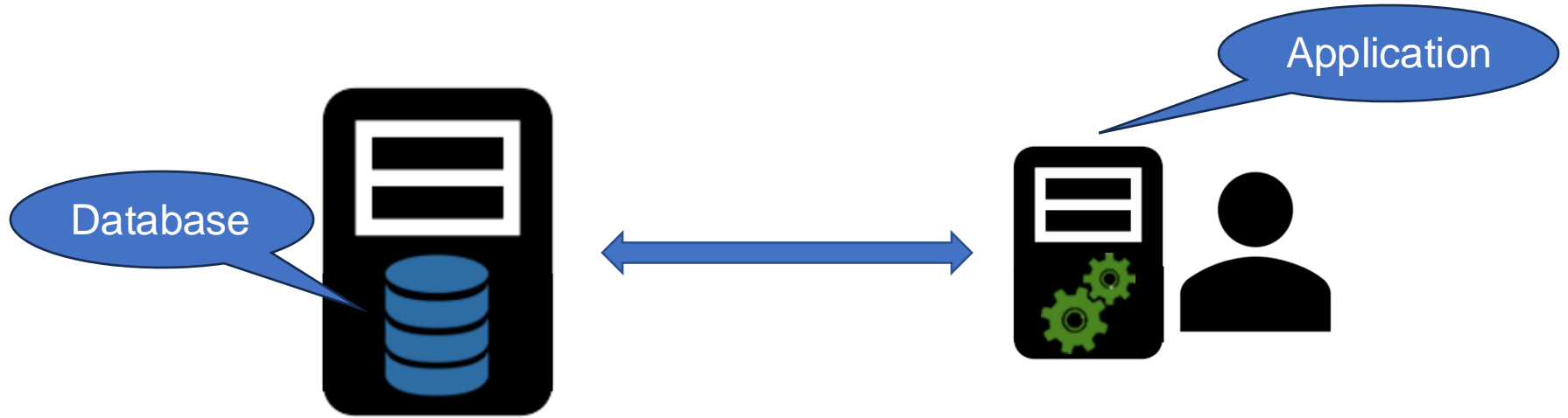
# Recap: SQL in a Programming Language

Acc	Usr	Balance
	Alice	300
	Bob	600
	Carol	400

```
...  
b1 = b+a          # the new balance  
cur.execute("UPDATE acc  
             SET balance = ?  
             WHERE usr=?",  
            [b1,usr])
```

# Recap: Single User

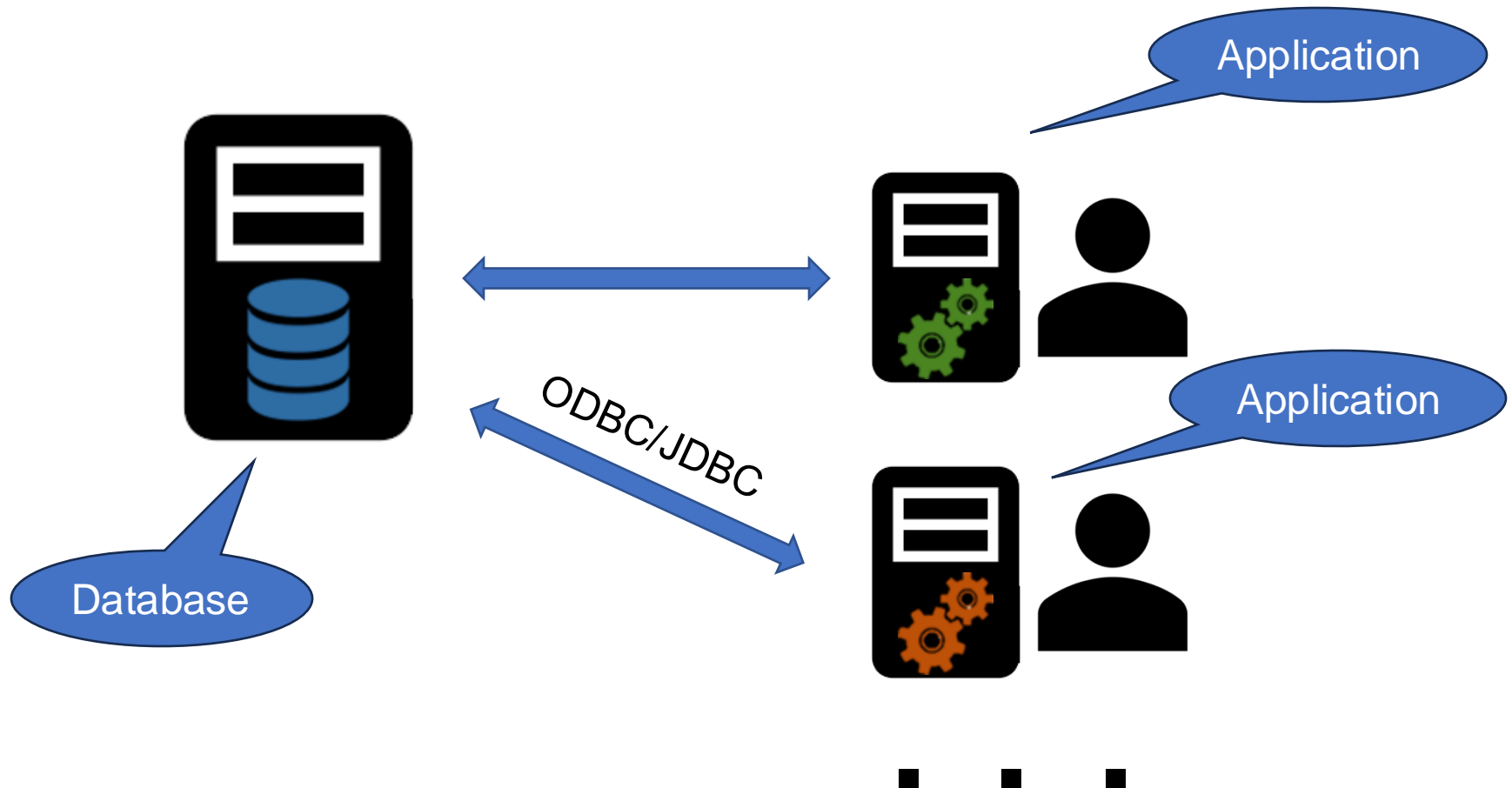
- The database is accessed by a single user:



- RDBMS on same laptop, or a server, or the cloud

# Recap: Client-Server

- Multiple users access the database concurrently



# Transactions

# Transactions

- A transaction is a set of read and writes to the database that execute all or nothing

**BEGIN TRANSACTION**

...SQL Statements

**COMMIT**

Entire txn is executed

**BEGIN TRANSACTION**

...SQL Statements

**ROLLBACK**

No part of txn is executed



# Transactions

- Prevent all concurrency control conflicts
- Easy to use in app: group statements in txns
- What property does a TXN satisfy?
  - Informally: “TXNs have ACID properties”
  - Formally: “execution of TXNs must be serializable”

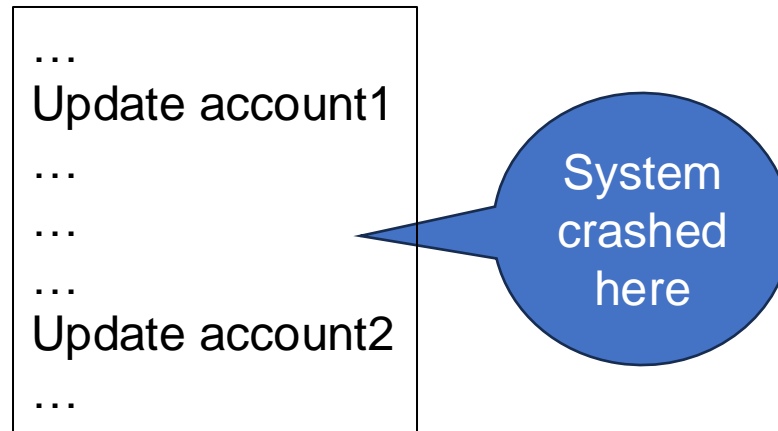
# ACID

# Transactions are ACID

- Atomic
- Consistent
- Isolated
- Durable

# Atomic

- A set of operations is atomic if either all its operations happen, or none happens



Recovery manager (not discussed in this class)

# Consistent

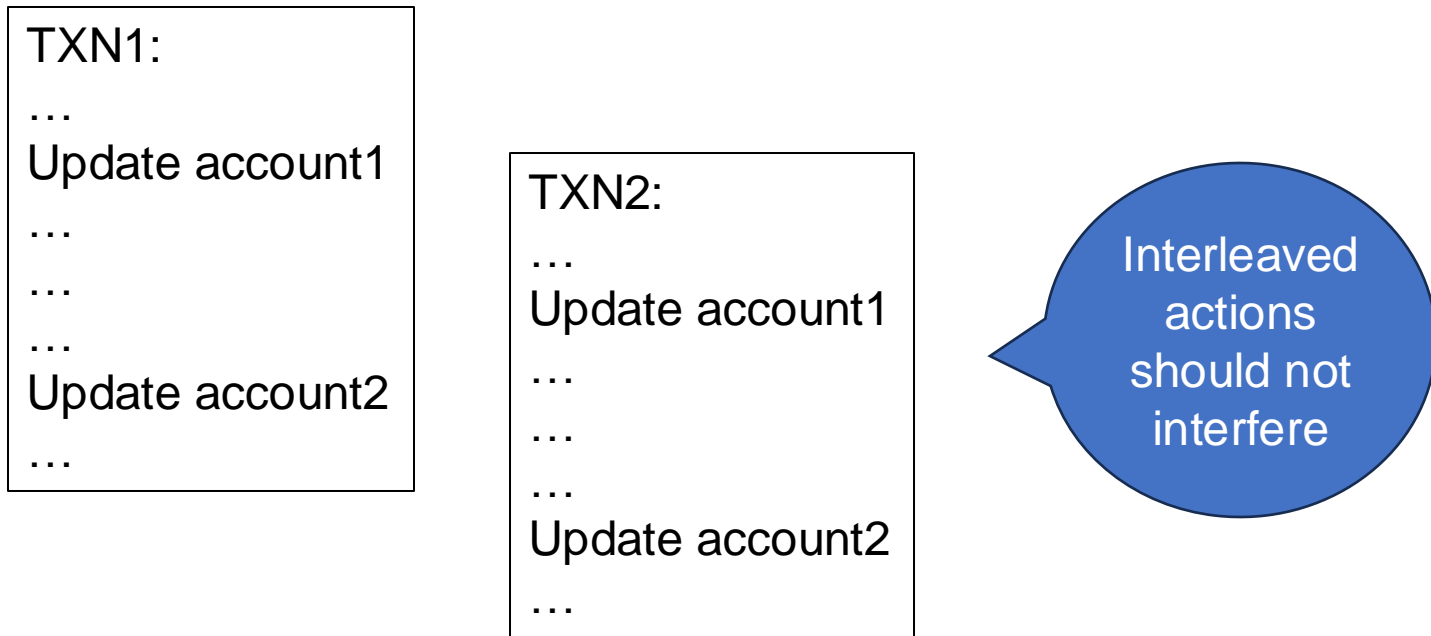
Assume TXN is “correct” (this is app specific)

- If TXN starts with the DB in a consistent state, it must end leaving the DB in a consistent state

It is a consequence of Atomicity and Isolation

# Isolated

- The effect of the transaction on the database is as if it were running alone on the database



Concurrency Control Manager

- Data should be stored persistently on disk, always in a consistent state

# Discussion

- ACID properties: popular job interview question

- “A” and “I” matter

- **Atomicity**: recover from crashes
- **Isolation**: concurrency control



444



344 and 444

- ACID is informal.

Will discuss the formal property next



# Serializability

# Problem Definition


- The RDBMs runs several TXNs: T1, T2, T3, ...
- It could run T1 to completion before starting T2, then run T2 to completion before starting T3, then run T3...  
...

# Problem Definition

- The RDBMs runs several TXNs: T1, T2, T3, ...
- It could run T1 to completion before starting T2, then run T2 to completion before starting T3, then run T3...

...

But this has poor performance




Why?  
(in class)

# Problem Definition

- The RDBMs runs several TXNs: T1, T2, T3, ...
- It could run T1 to completion before starting T2, then run T2 to completion before starting T3, then run T3...

...

But this has poor performance

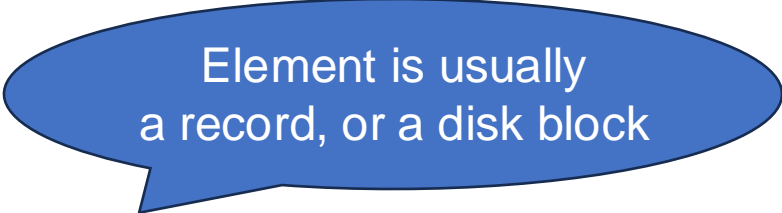


Why?  
(in class)

- Instead: interleave commands from multiple TXNs

When is the interleaving "safe"?

# Simplified Data Model for TXN



Element is usually  
a record, or a disk block

- Database = a set of “elements”
- TXN = a sequence of Reads/Writes of elements

# Example

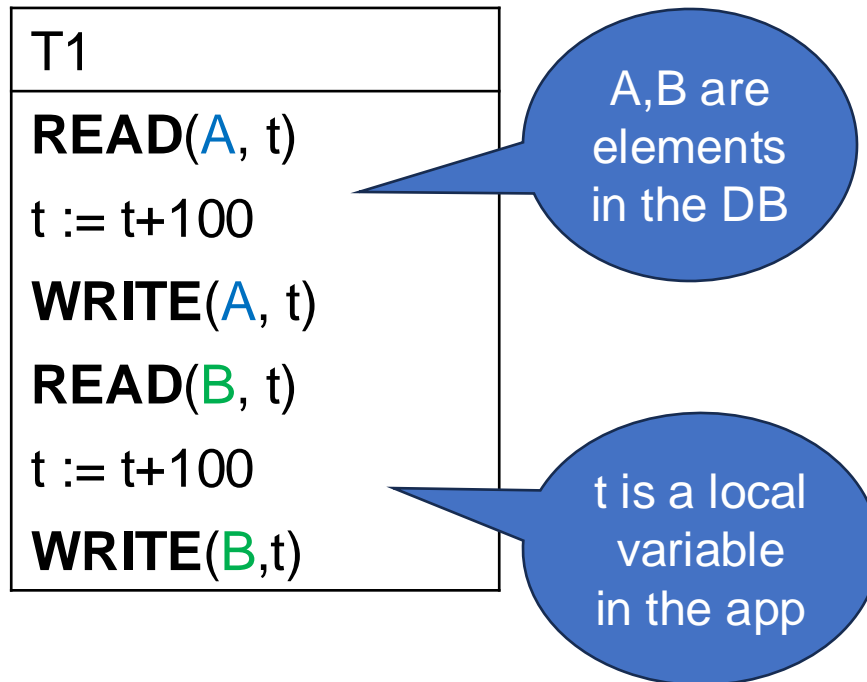
T1
<b>READ</b> (A, t) t := t+100 <b>WRITE</b> (A, t) <b>READ</b> (B, t) t := t+100 <b>WRITE</b> (B,t)

# Example

T1
<b>READ</b> (A, t)
t := t+100
<b>WRITE</b> (A, t)
<b>READ</b> (B, t)
t := t+100
<b>WRITE</b> (B,t)

A,B are  
elements  
in the DB

# Example





# Example

T1
<b>READ</b> (A, t)
t := t+100
<b>WRITE</b> (A, t)
<b>READ</b> (B, t)
t := t+100
<b>WRITE</b> (B,t)

A,B are  
elements  
in the DB

t is a local  
variable  
in the app

T2
<b>READ</b> (A, s)
s := s*2
<b>WRITE</b> (A,s)
<b>READ</b> (B,s)
s := s*2
<b>WRITE</b> (B,s)

# Definitions

- An interleaving of READ/WRITEs from different TXNs is called a **schedule**

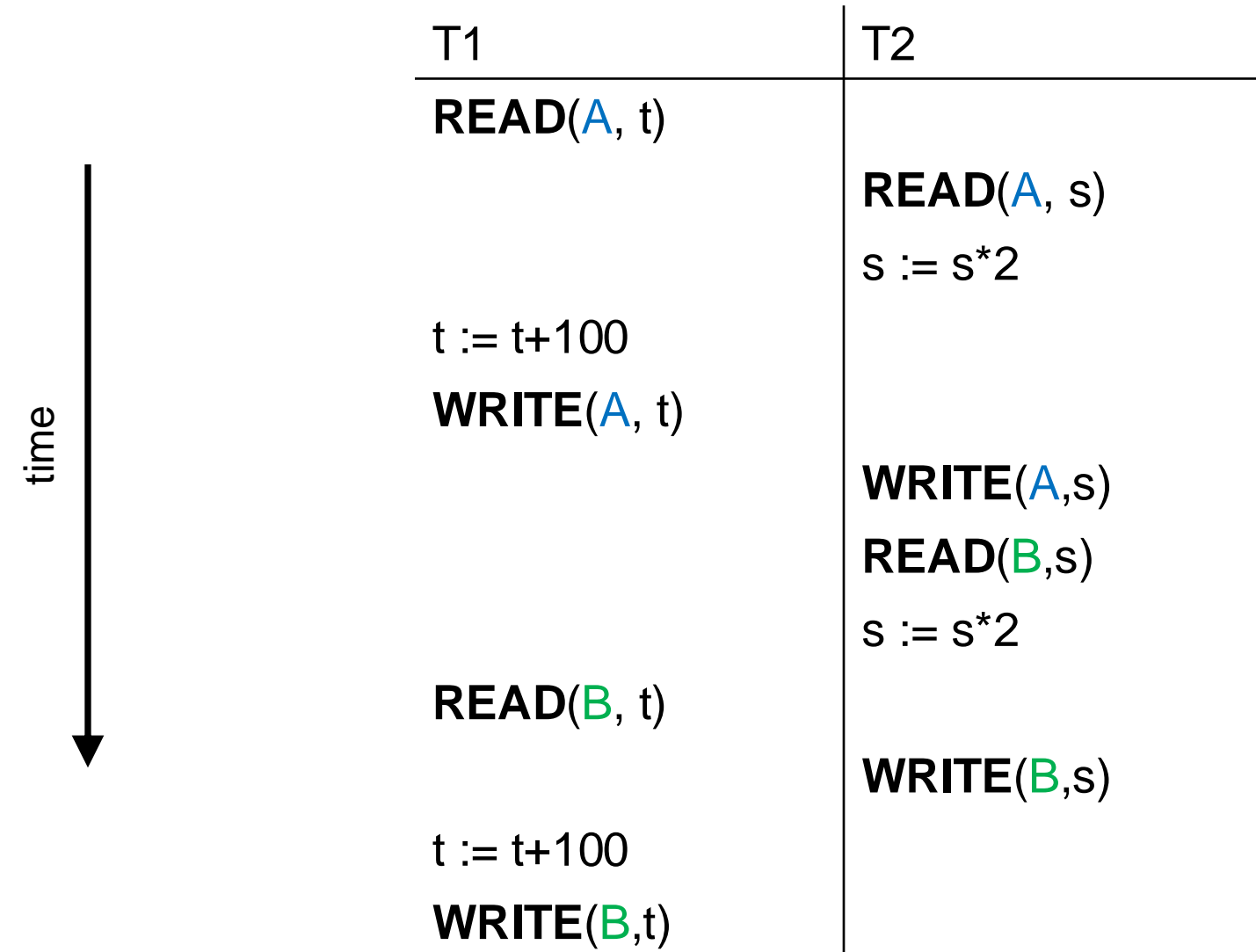
# Definitions

- An interleaving of READ/WRITEs from different TXNs is called a **schedule**
- **Definition:** a **serial schedule** is a schedule where all operations of transactions come before those of the next transaction

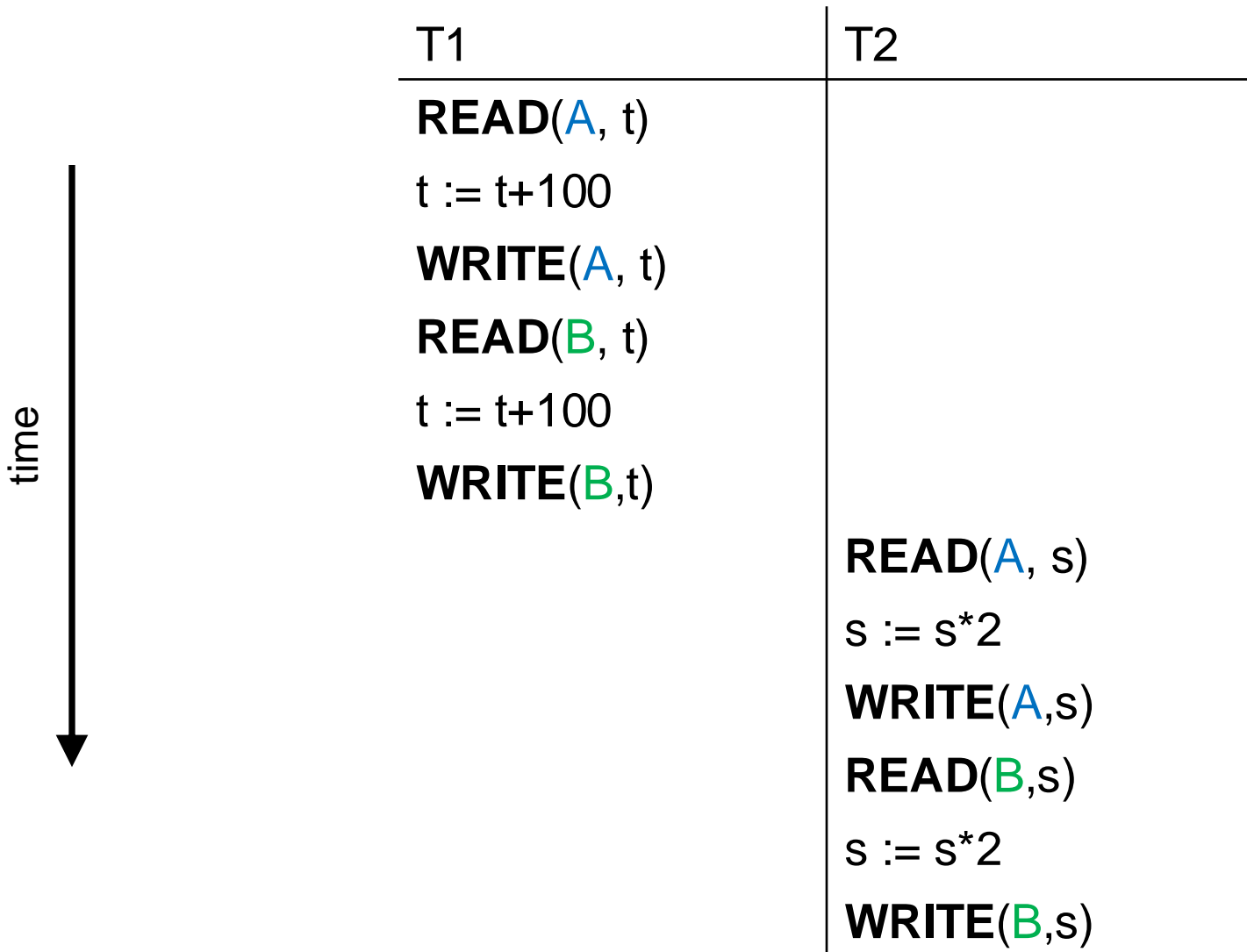
# Definitions

- An interleaving of READ/WRITEs from different TXNs is called a **schedule**
- **Definition:** a **serial schedule** is a schedule where all operations of transactions come before those of the next transaction
- **Definition:** a **serializable schedule** is a schedule that is equivalent to a serial schedule

# A Schedule



# A Serial Schedule



# A Serial Schedule

A = 2  
B = 2

time  
↓

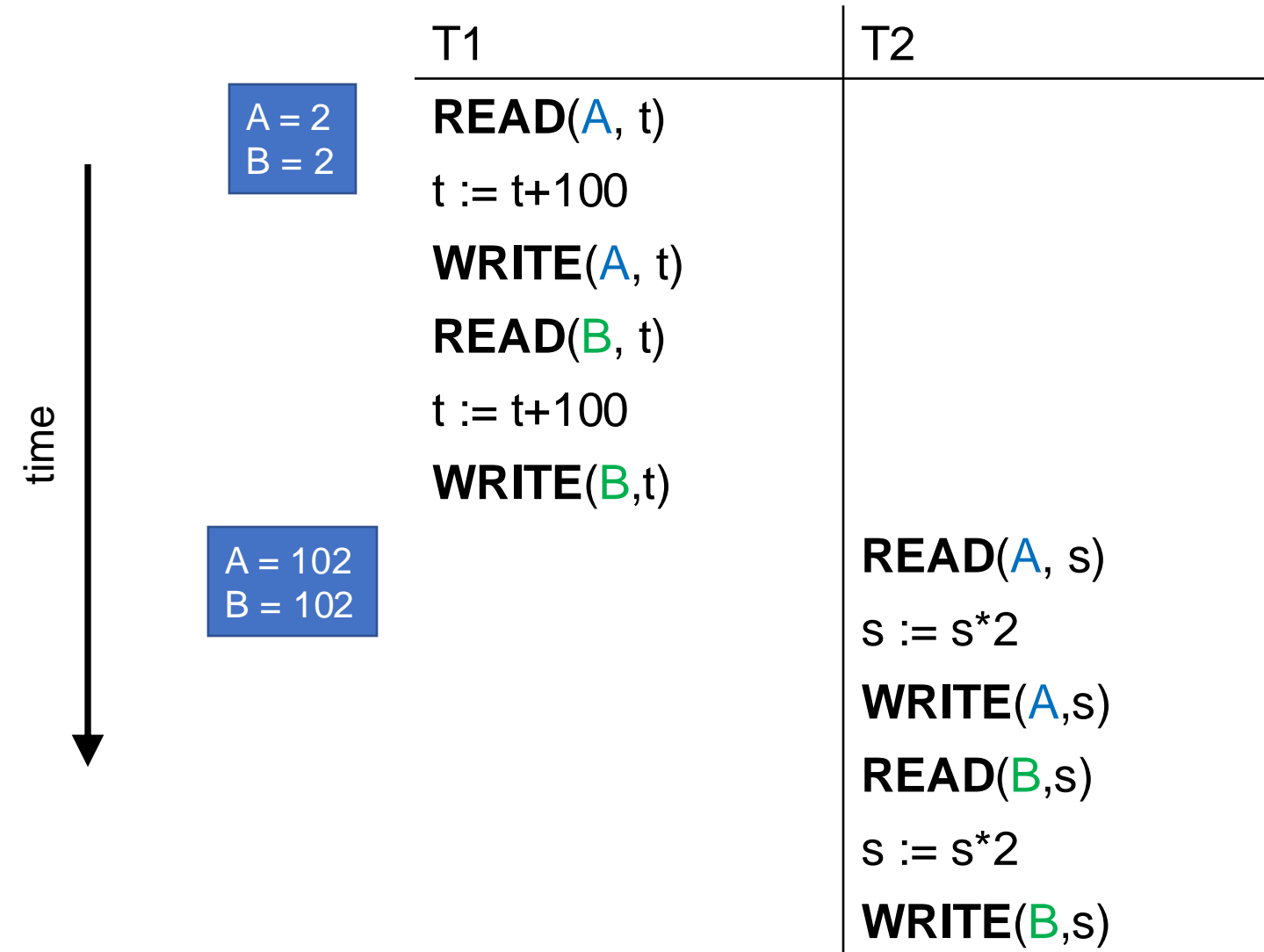
T1

**READ**(A, t)  
t := t+100  
**WRITE**(A, t)  
**READ**(B, t)  
t := t+100  
**WRITE**(B, t)

T2

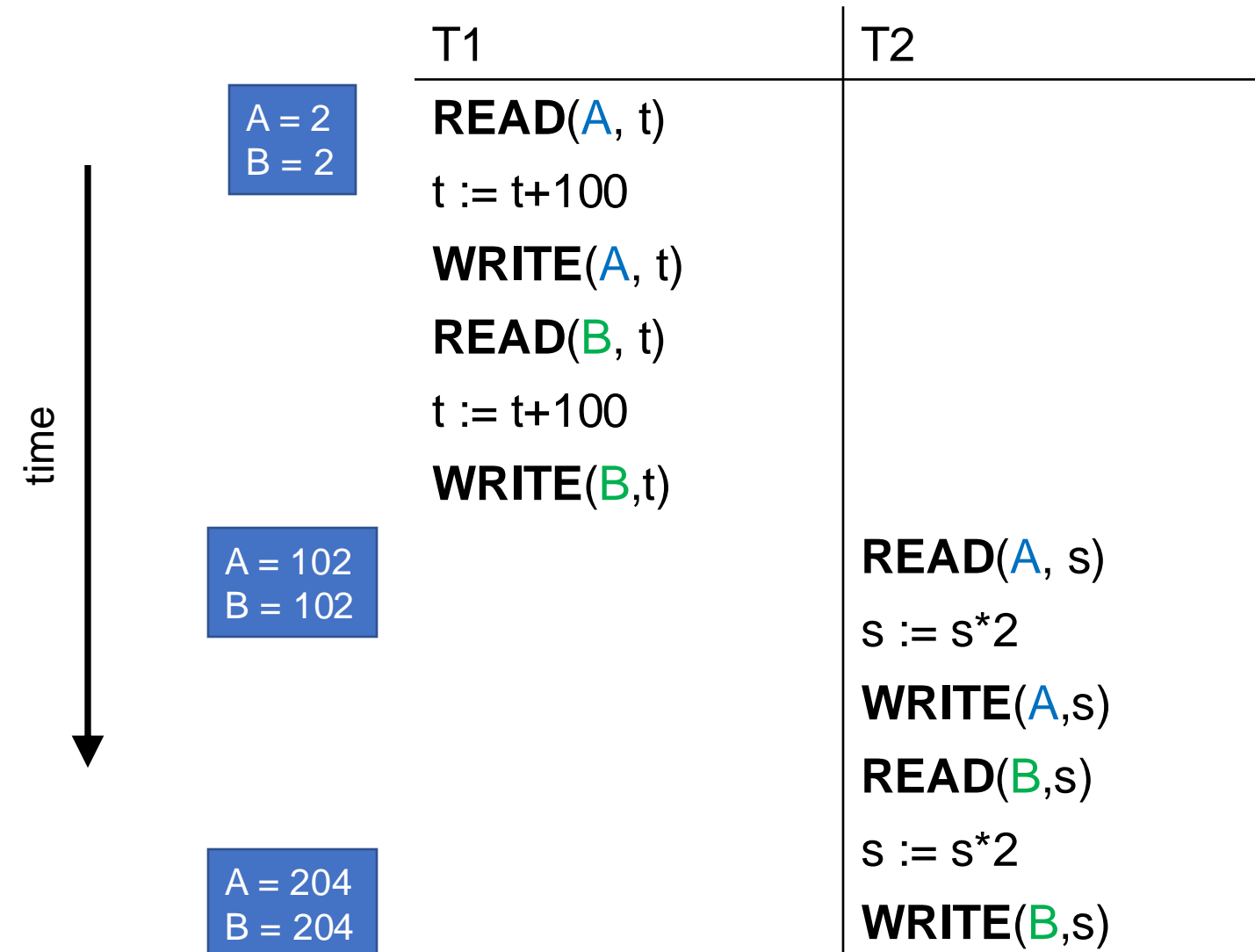
**READ**(A, s)  
s := s\*2  
**WRITE**(A, s)  
**READ**(B, s)  
s := s\*2  
**WRITE**(B, s)

# A Serial Schedule

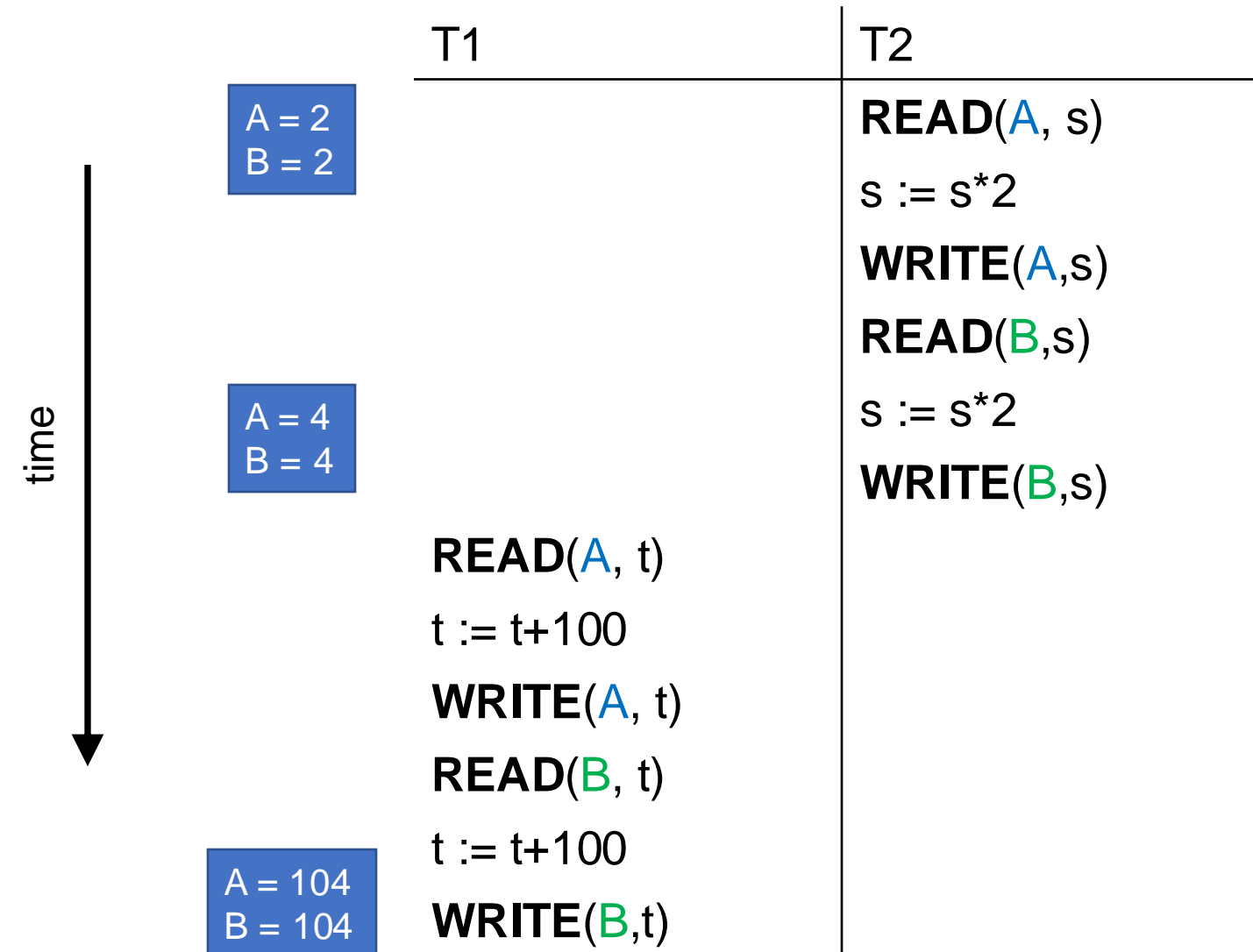




# A Serial Schedule



# The Other **Serial** Schedule



# A Serializable Schedule

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

# A Serializable Schedule

T1	T2	
<b>READ</b> (A, t)		A = 2 B = 2
t := t+100		
<b>WRITE</b> (A, t)		
	<b>READ</b> (A, s)	A = 102 B = 2
	s := s*2	
	<b>WRITE</b> (A,s)	
<b>READ</b> (B, t)		
t := t+100		
<b>WRITE</b> (B,t)		
	<b>READ</b> (B,s)	
	s := s*2	
	<b>WRITE</b> (B,s)	

# A Serializable Schedule

T1	T2	
<b>READ</b> (A, t)		A = 2 B = 2
t := t+100		
<b>WRITE</b> (A, t)		
	<b>READ</b> (A, s)	A = 102 B = 2
	s := s*2	
	<b>WRITE</b> (A,s)	A = 204 B = 2
<b>READ</b> (B, t)		
t := t+100		
<b>WRITE</b> (B,t)		
	<b>READ</b> (B,s)	
	s := s*2	
	<b>WRITE</b> (B,s)	

# A Serializable Schedule

T1	T2	
<b>READ</b> (A, t)		A = 2 B = 2
t := t+100		
<b>WRITE</b> (A, t)		
	<b>READ</b> (A, s)	A = 102 B = 2
	s := s*2	
	<b>WRITE</b> (A,s)	A = 204 B = 2
<b>READ</b> (B, t)		
t := t+100		
<b>WRITE</b> (B,t)		A = 204 B = 102
	<b>READ</b> (B,s)	
	s := s*2	A = 204 B = 204
	<b>WRITE</b> (B,s)	

# A Serializable Schedule

T1	T2	
<b>READ</b> (A, t)		A = 2 B = 2
t := t+100		
<b>WRITE</b> (A, t)		
	<b>READ</b> (A, s)	A = 102 B = 2
	s := s*2	
	<b>WRITE</b> (A,s)	A = 204 B = 2
<b>READ</b> (B, t)		
t := t+100		
<b>WRITE</b> (B,t)		
	<b>READ</b> (B,s)	A = 204 B = 102
	s := s*2	
	<b>WRITE</b> (B,s)	A = 204 B = 204

This is NOT a serial schedule  
It is a serializable schedule.

# A Non-Serializable Schedule

T1	T2	
<b>READ</b> (A, t)		A = 2 B = 2
t := t+100		
<b>WRITE</b> (A, t)		
	<b>READ</b> (A, s)	A = 102 B = 2
	s := s*2	
	<b>WRITE</b> (A,s)	
	<b>READ</b> (B,s)	
	s := s*2	
	<b>WRITE</b> (B,s)	
<b>READ</b> (B, t)		
t := t+100		
<b>WRITE</b> (B,t)		



# A Non-Serializable Schedule

T1	T2	
<b>READ</b> (A, t)		A = 2 B = 2
t := t+100		
<b>WRITE</b> (A, t)		
	<b>READ</b> (A, s)	A = 102 B = 2
	s := s*2	
	<b>WRITE</b> (A,s)	A = 204 B = 2
	<b>READ</b> (B,s)	
	s := s*2	
	<b>WRITE</b> (B,s)	
<b>READ</b> (B, t)		
t := t+100		
<b>WRITE</b> (B,t)		

# A Non-Serializable Schedule

T1	T2	
<b>READ</b> (A, t)		A = 2 B = 2
t := t+100		
<b>WRITE</b> (A, t)		
	<b>READ</b> (A, s)	A = 102 B = 2
	s := s*2	
	<b>WRITE</b> (A,s)	A = 204 B = 2
	<b>READ</b> (B,s)	
	s := s*2	
	<b>WRITE</b> (B,s)	A = 204 B = 4
<b>READ</b> (B, t)		
t := t+100		
<b>WRITE</b> (B,t)		

# A Non-Serializable Schedule

T1	T2	
<b>READ</b> (A, t)		A = 2 B = 2
t := t+100		
<b>WRITE</b> (A, t)		
	<b>READ</b> (A, s)	A = 102 B = 2
	s := s*2	
	<b>WRITE</b> (A,s)	A = 204 B = 2
	<b>READ</b> (B,s)	
	s := s*2	
	<b>WRITE</b> (B,s)	A = 204 B = 4
<b>READ</b> (B, t)		
t := t+100		
<b>WRITE</b> (B,t)		A = 204 B = 104

# A Non-Serializable Schedule

T1	T2	
<b>READ</b> (A, t)		A = 2 B = 2
t := t+100		
<b>WRITE</b> (A, t)		A = 102 B = 2
	<b>READ</b> (A, s)	
	s := s*2	
	<b>WRITE</b> (A,s)	A = 204 B = 2
	<b>READ</b> (B,s)	
	s := s*2	
	<b>WRITE</b> (B,s)	A = 204 B = 4
<b>READ</b> (B, t)		
t := t+100		
<b>WRITE</b> (B,t)	Should be impossible!	A = 204 B = 104

# Discussion

- If the schedule is serial, then nothing can go wrong
- Same for a serializable schedule
- Concurrency Control Manager of the RDBMs must ensure that the schedule is serializable

How do we check that a schedule is serializable?

# Conflict Serializability

We further simplify the model:

- A transaction is a sequence of reads and writes
- We ignore operations between reads and writes

# Example

T1
<b>READ</b> (A, t)
t := t+100
<b>WRITE</b> (A, t)
<b>READ</b> (B, t)
t := t+100
<b>WRITE</b> (B,t)



T1
R(A)
W(A)
R(B)
W(B)

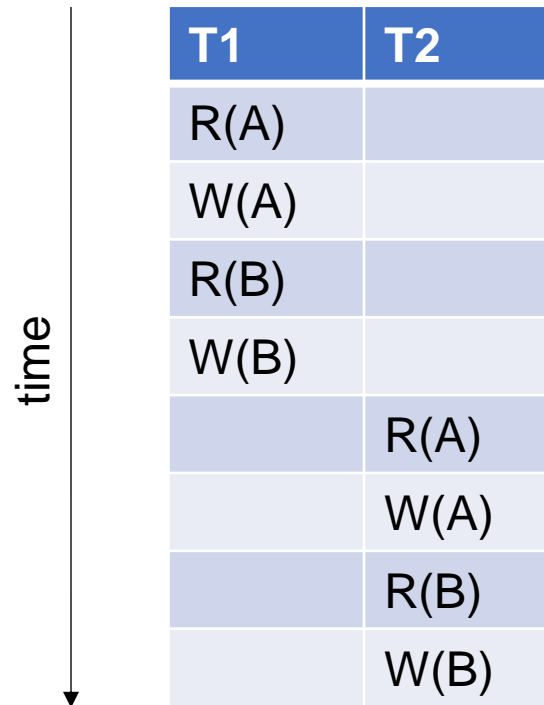
Also:  $R_1(A), W_1(A), R_1(B), W_1(B)$



# Example

- T1 then T2

$R_1(A)$ ,  $W_1(A)$ ,  $R_1(B)$ ,  $W_1(B)$ ,  $R_2(A)$ ,  $W_2(A)$ ,  $R_2(B)$ ,  $W_2(B)$



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

# Example

- T2 then T1

$R_2(A), W_2(A), R_2(B), W_2(B), R_1(A), W_1(A), R_1(B), W_1(B)$

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

# Example

- Serializable to T1 then T2

$R_1(A), W_1(A), R_2(A), W_2(A), R_1(B), W_1(B), R_2(B), W_2(B)$

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

# Example

- Not serializable

$R_1(A), W_1(A), R_2(A), W_2(A), R_2(B), W_2(B), R_1(B), W_1(B)$

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

# Main Idea

- To check if a schedule is serializable, try swapping operations until it becomes serial:



- But we only swap if the new schedule is equivalent
- A pair is in **conflict** if it cannot be swapped

# Conflicts

1. Any pair of ops of the same TXN are in conflict
2.  $R_i(X), W_j(X)$  forms a read-write conflict
3.  $W_i(X), R_j(X)$  forms a write-read conflict
4.  $W_i(X), W_j(X)$  forms a write-write conflict

# Conflict Serializable Schedule

A schedule is conflict serializable if it can be **transformed** into a serial schedule by a series of swappings of adjacent **non-conflicting** actions

# Conflict Serializable Schedule Example

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



# Conflict Serializable Schedule Example

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

# Conflict Serializable Schedule Example

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

# Conflict Serializable Schedule Example

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

# Conflict Serializable Schedule Example

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

# Non Conflict Serializable Schedule Example

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

# Non Conflict Serializable Schedule Example

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



Conflict rule broken!

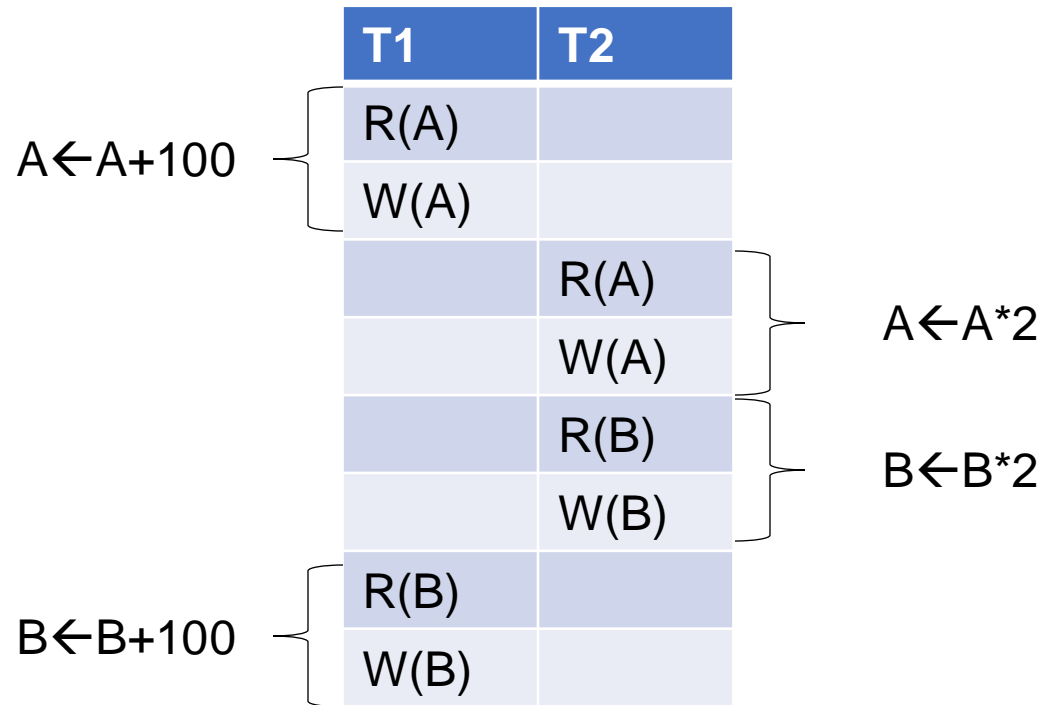
# Serializable vs Conflict Serializable

Conflict serializability ignores what TXN does between the R's and the W's.  
It assumes the worst / most complicated updates to the data

# Serializable vs Conflict Serializable

Conflict serializability ignores what TXN does between the R's and the W's.  
It assumes the worst / most complicated updates to the data

Not serializable nor conflict serializable



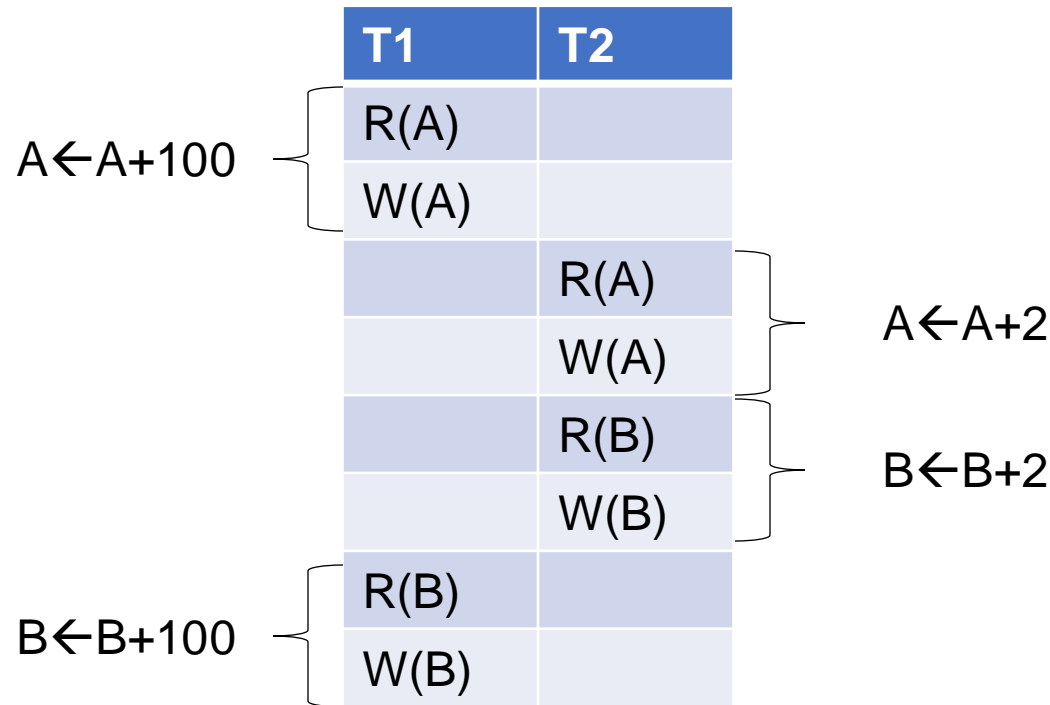


# Serializable vs Conflict Serializable

Conflict serializability ignores what TXN does between the R's and the W's.  
It assumes the worst / most complicated updates to the data

Serializable (because  $100+2 = 2+100$ )

But not conflict serializable, because it assumes the worst



- Most RDBMs enforce conflict-serializability
- Next: how to test for conflict-serializability

# The Precedence Graph

# Testing for Conflict Serializability

Fix a schedule

- **Definition.** The **precedence graph** has one node for every TXN in the schedule, and one edge for every pair of conflicting ops
- **Theorem.** The schedule is conflict-serializable iff the precedence graph has no cycles

# Example 1

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

# Example 1

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

Nodes:

①

②

③

# Example 1

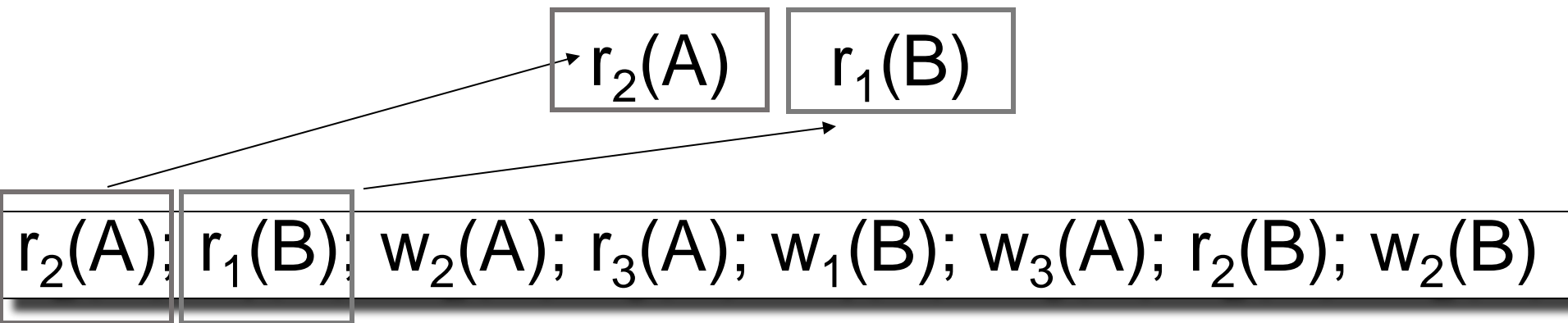
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

Edges:

①

②

③



Edges:

①

②

③



$r_2(A)$   $r_1(B)$

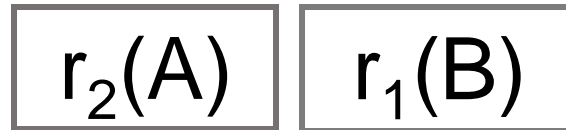
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

Edges:

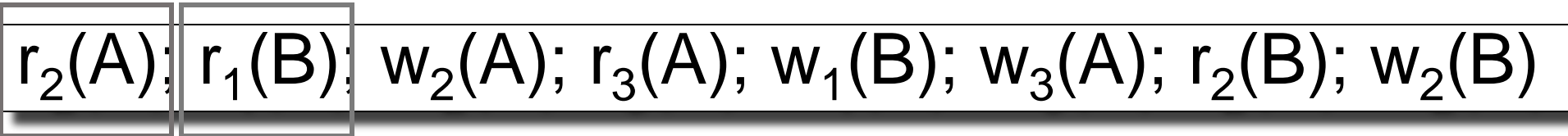
①

②

③



No edge because  
no conflict ( $A \neq B$ )



Edges:

①

②

③

$r_2(A)$   $w_2(A)$

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

Edges:

①

②

③

$r_2(A)$   $w_2(A)$

No edge because  
same txn (2)

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

Edges:

①

②

③

$r_2(A)$   $r_3(A)$  ?

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

Edges:

①

②

③

$r_2(A)$   $w_1(B)$  ?

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

Edges:

①

②

③

$r_2(A)$   $w_3(A)$  ?

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

Edges:

①

②

③

$r_2(A)$   $w_3(A)$

Edge! Conflict from  
T2 to T3

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

Edges:

①

②

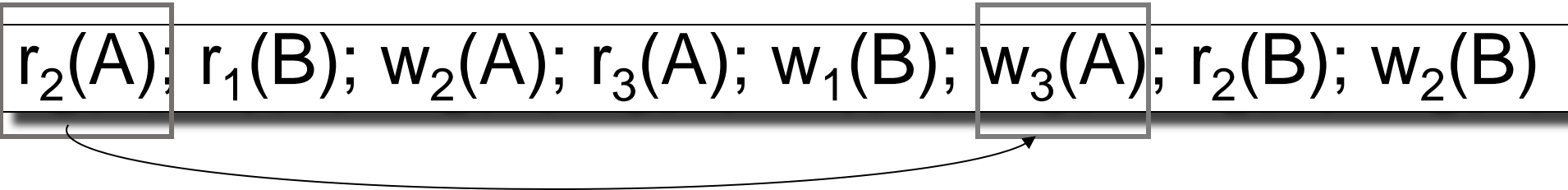
③



$r_2(A)$     $w_3(A)$

Edge! Conflict from  
T2 to T3

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



Edges:



$r_2(A)$   $r_2(B)$  ?

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

Edges:

①

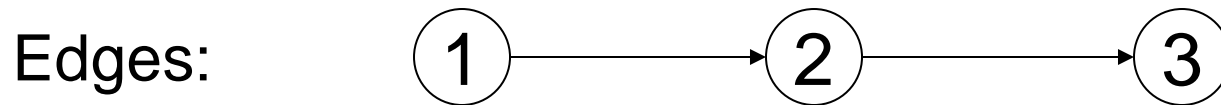
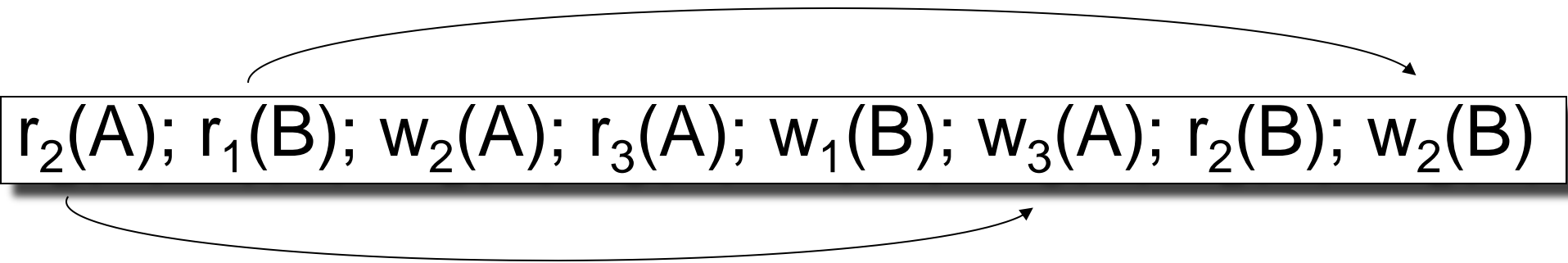
②

A

③

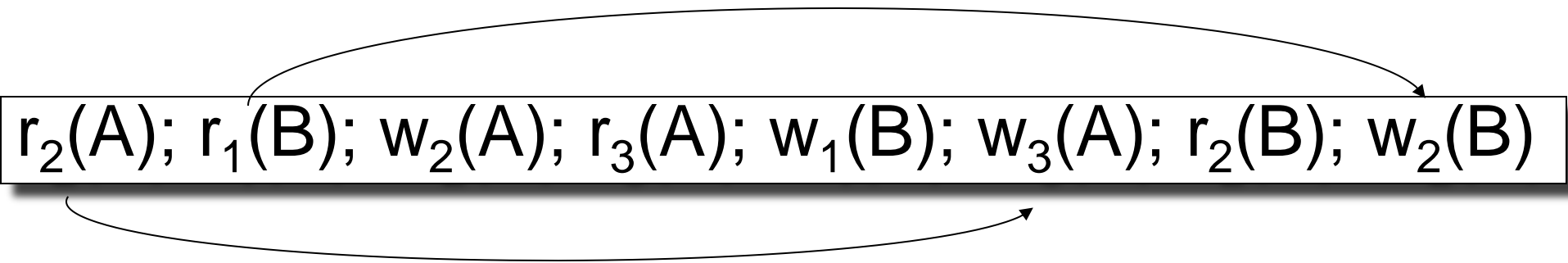
And so on until compared every pair of actions...

# Example 1



Repeating the same directed edge not necessary

# Example 1



This schedule is **conflict-serializable**

# Example 2

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

# Example 2

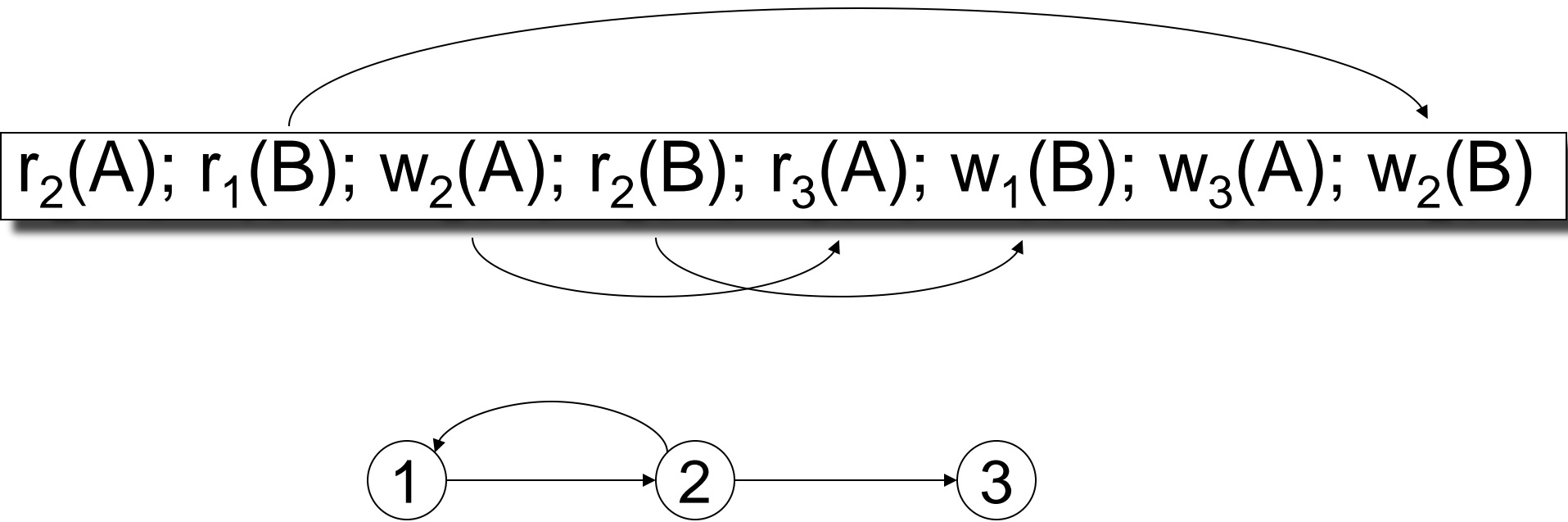
$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$

①

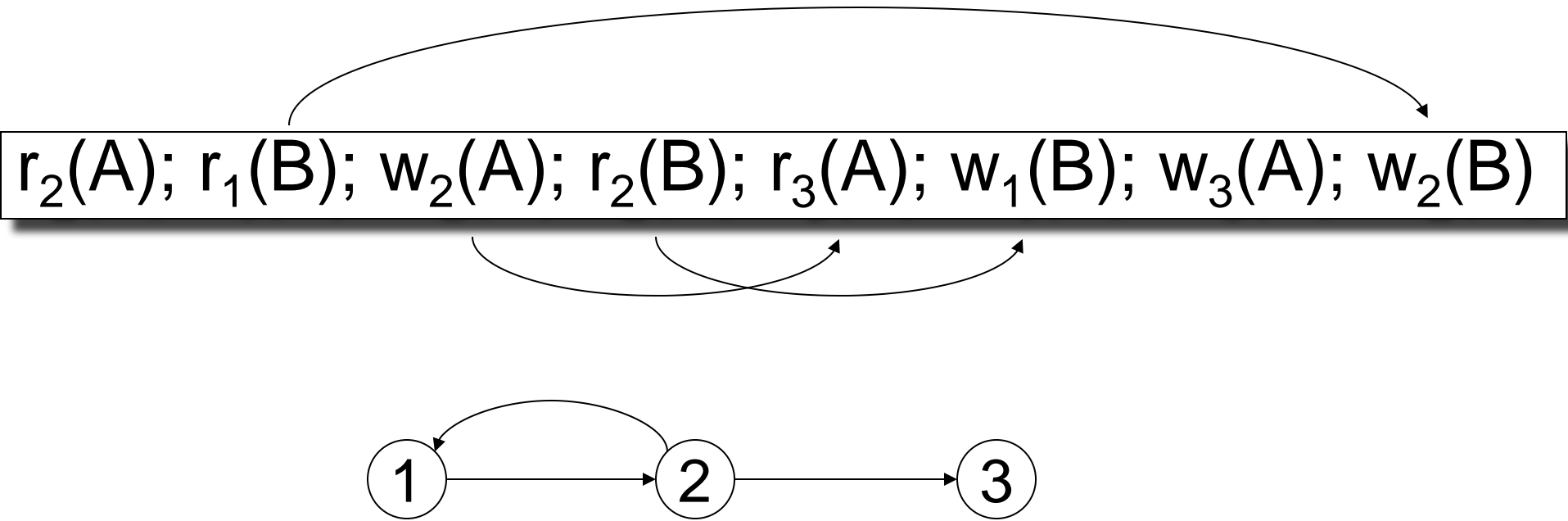
②

③

# Example 2



# Example 2



This schedule is NOT conflict-serializable



# Takeaways

- Transactions: “...all or nothing...”
- Simplified data model: READ/WRITE elements
- Schedules:
  - Serial
  - Serializable
  - Conflict serializable
- Precedence graph