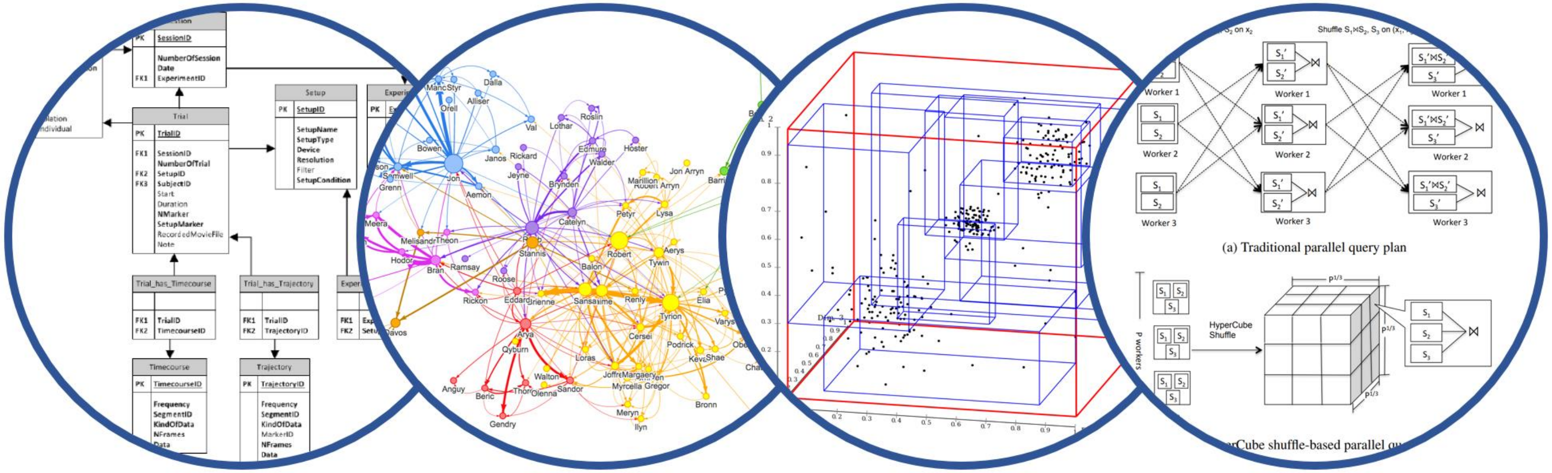


Introduction to Data Management

Transactions: Schedules

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle

Transactions

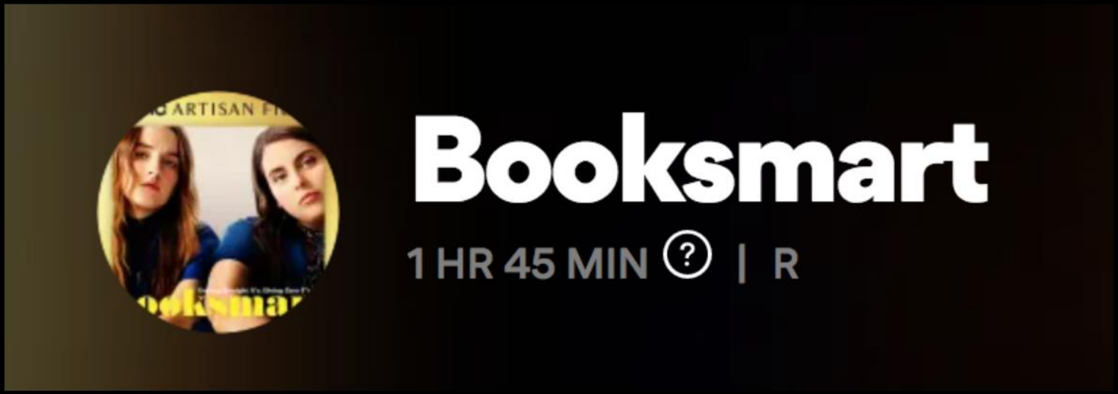
How do we support multiple people using a database at the same time?

- Multiple application users
- Multiple application programmers
- Multiple analysts
- Imagine a world where each person had to wait in line to use your database ☹

Common Concurrency Control Problems

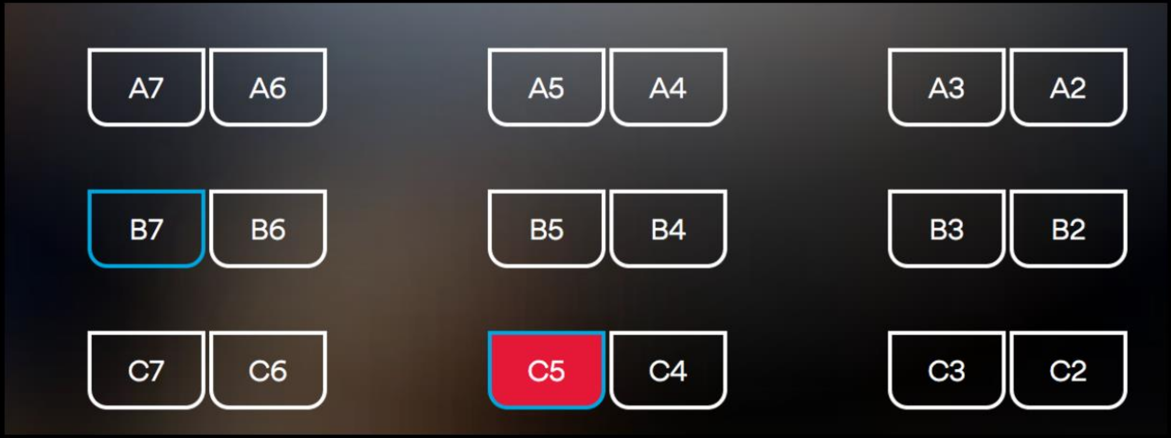
- Non-Atomic Operations
- Lost Update
- Dirty/Inconsistent Read
- Unrepeatable Read
- Phantom Read

Non-Atomic Operations

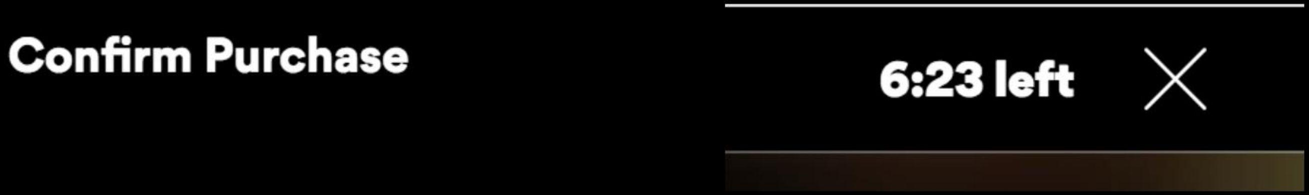


Booksmart

1 HR 45 MIN ? | R



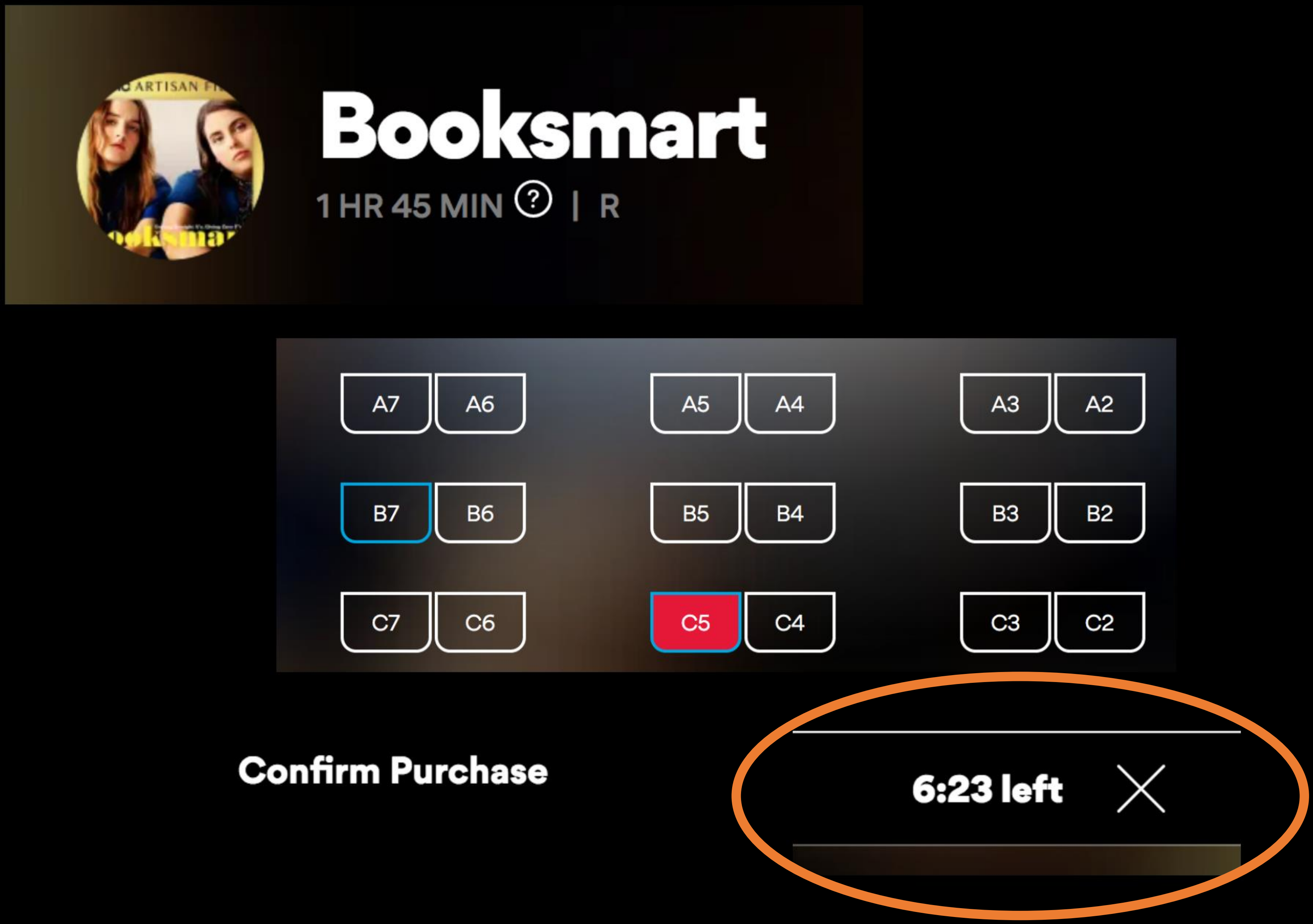
A7	A6	A5	A4	A3	A2
B7	B6	B5	B4	B3	B2
C7	C6	C5	C4	C3	C2



Confirm Purchase

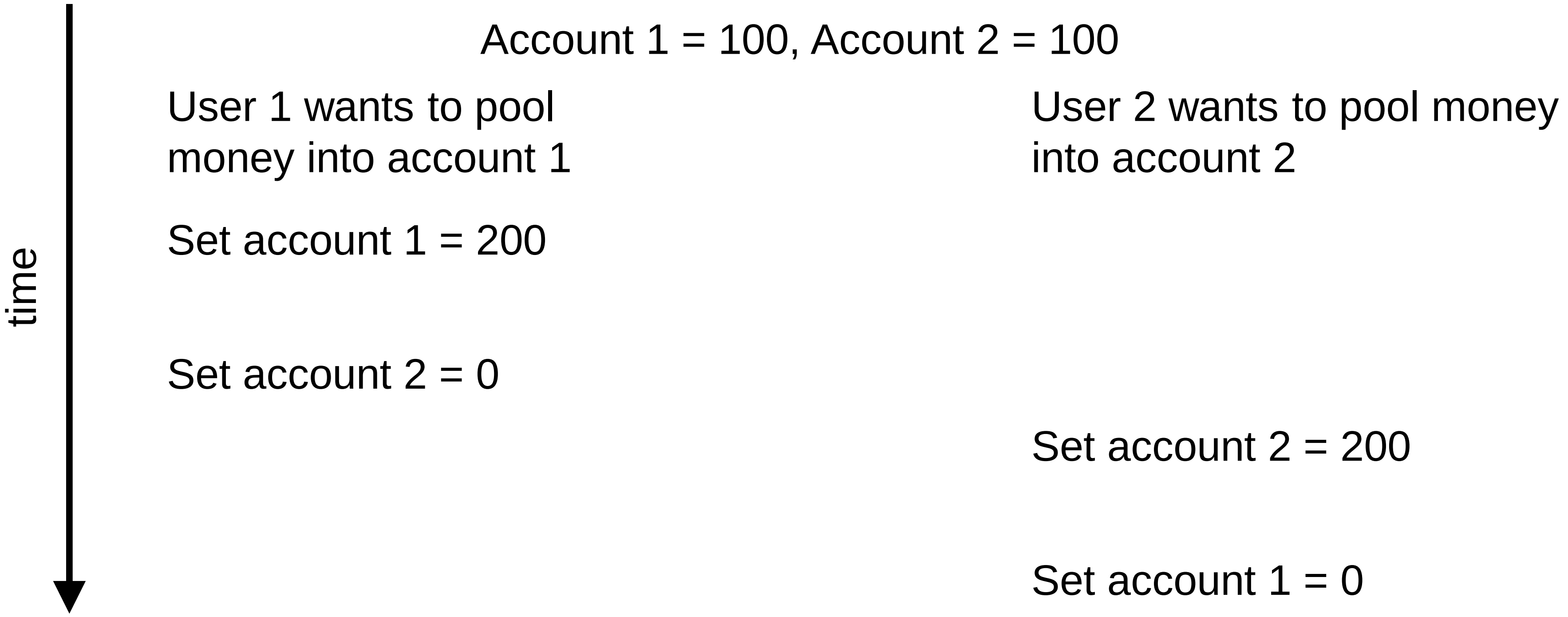
6:23 left ✕

Non-Atomic Operations



Lost Update

- Write-Write (WW) conflict
- Consolidation scenario:



Account 1 = 100, Account 2 = 100

User 1 wants to pool money into account 1

User 2 wants to pool money into account 2

Set account 1 = 200

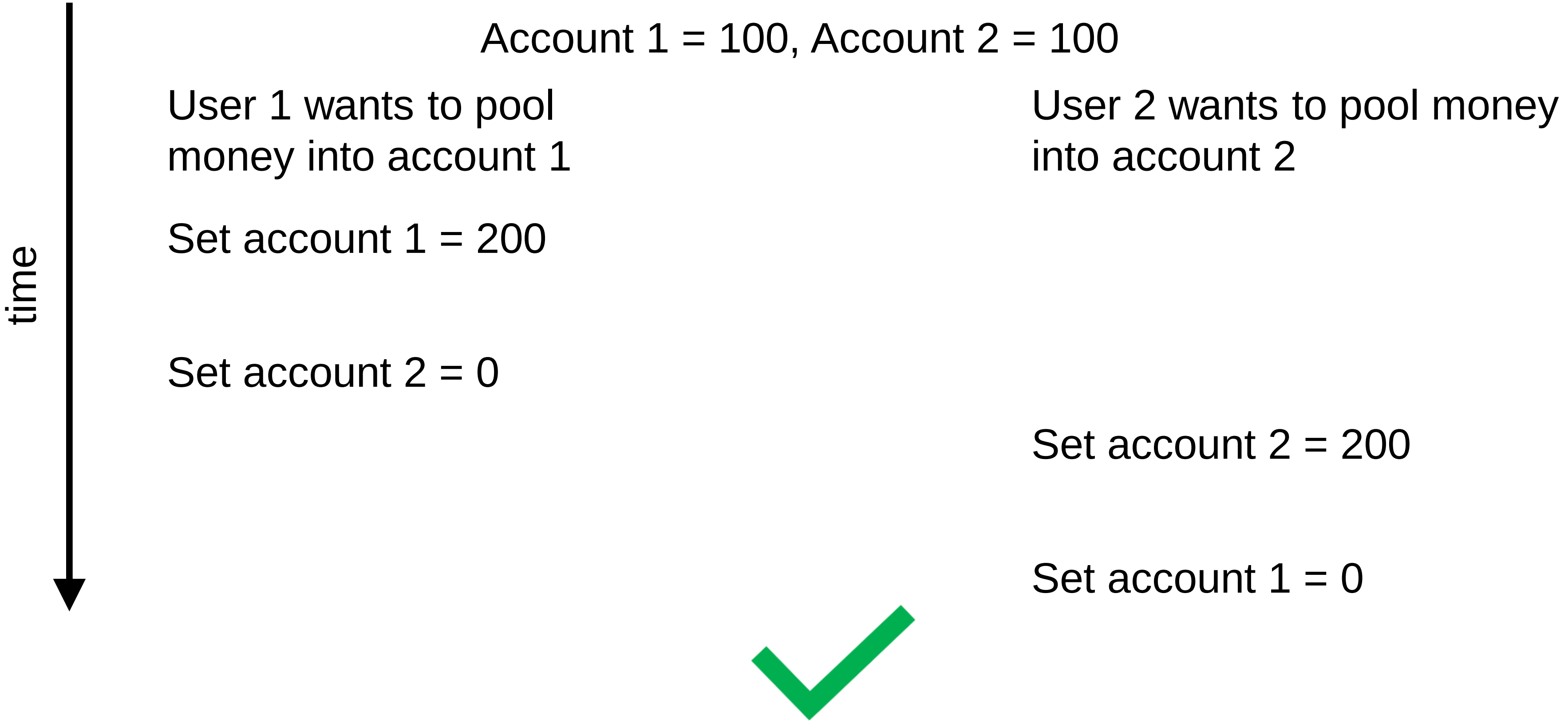
Set account 2 = 0

Set account 2 = 200

Set account 1 = 0

Lost Update

- Write-Write (WW) conflict
- Consolidation scenario:



Account 1 = 100, Account 2 = 100

User 1 wants to pool money into account 1

User 2 wants to pool money into account 2

Set account 1 = 200

Set account 2 = 0

Set account 2 = 200

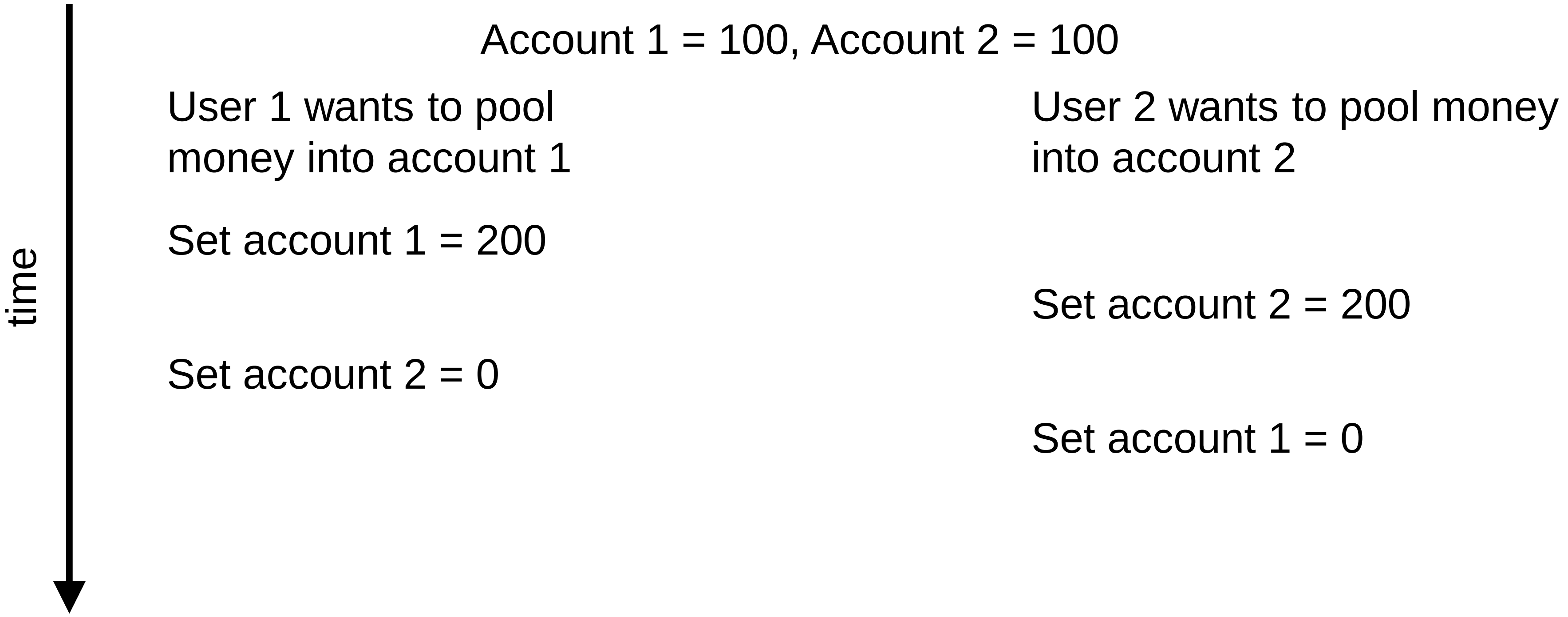
Set account 1 = 0

✓

At end: Account 1 = 0, Account 2 = 200

Lost Update

- Write-Write (WW) conflict
- Consolidation scenario:



Account 1 = 100, Account 2 = 100

User 1 wants to pool money into account 1

User 2 wants to pool money into account 2

Set account 1 = 200

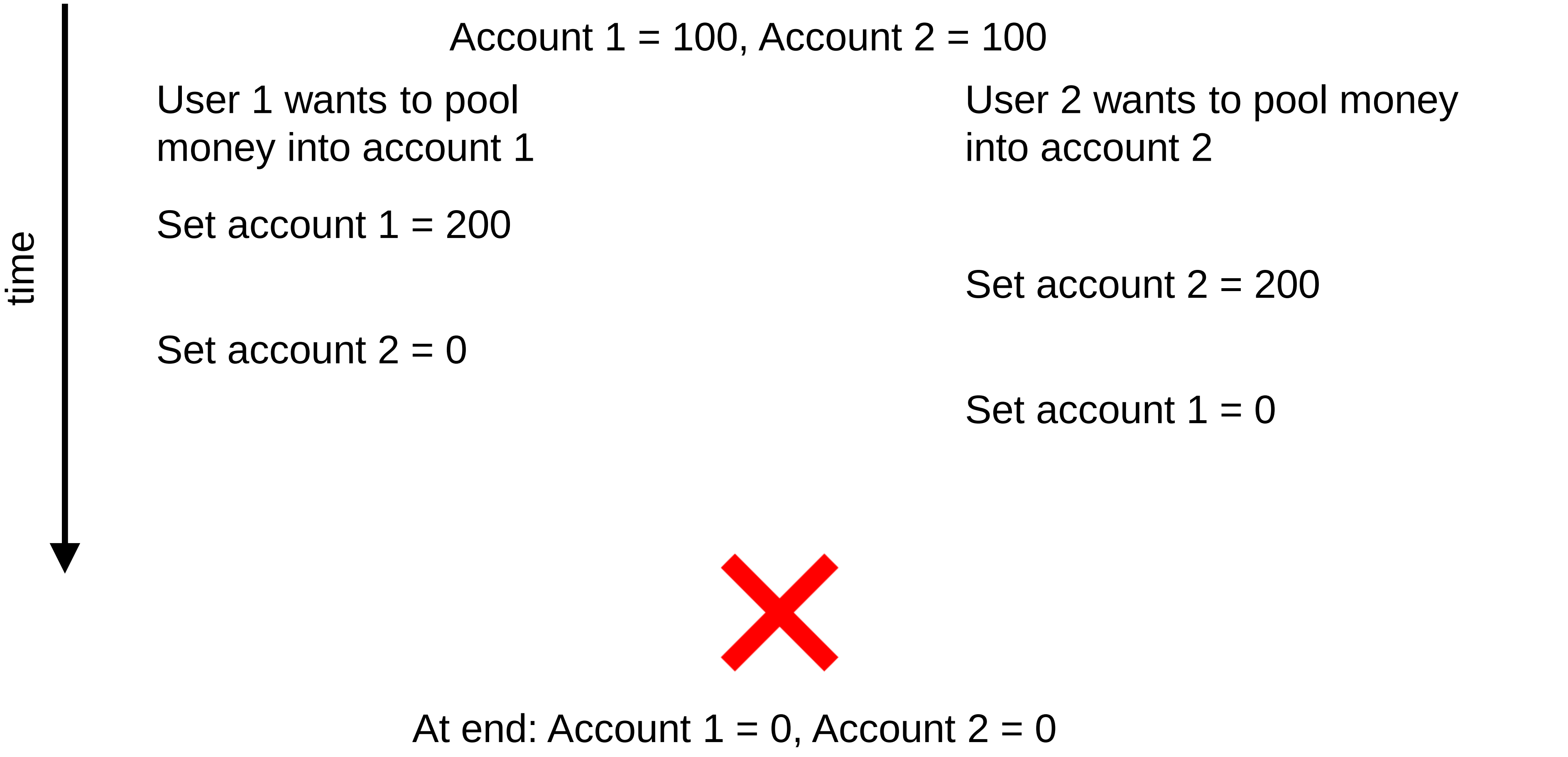
Set account 2 = 200

Set account 2 = 0

Set account 1 = 0

Lost Update

- Write-Write (WW) conflict
- Consolidation scenario:



Account 1 = 100, Account 2 = 100

User 1 wants to pool money into account 1

User 2 wants to pool money into account 2

Set account 1 = 200

Set account 2 = 200

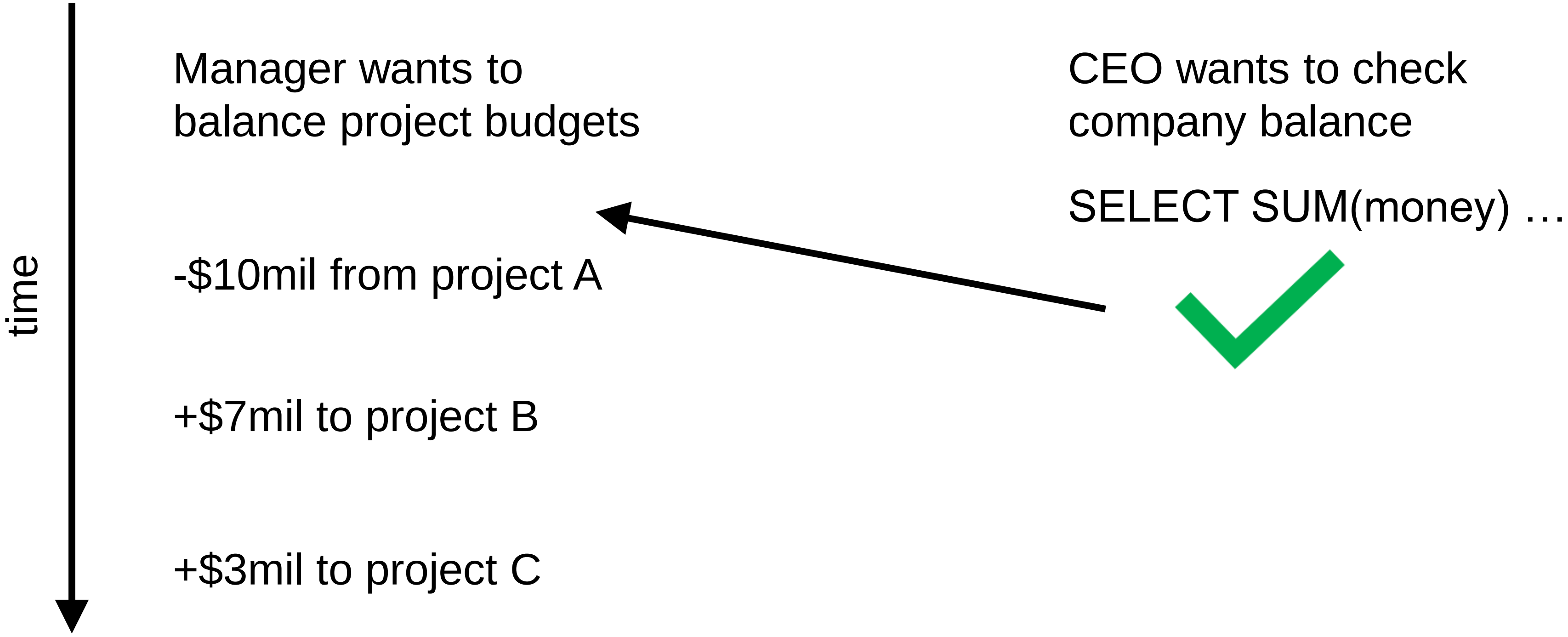
Set account 2 = 0

Set account 1 = 0

At end: Account 1 = 0, Account 2 = 0

Dirty/Inconsistent Read

- Write-Read (WR) conflict
- Budget management scenario:



time

Manager wants to balance project budgets

-$10mil from project A

+$7mil to project B

+$3mil to project C

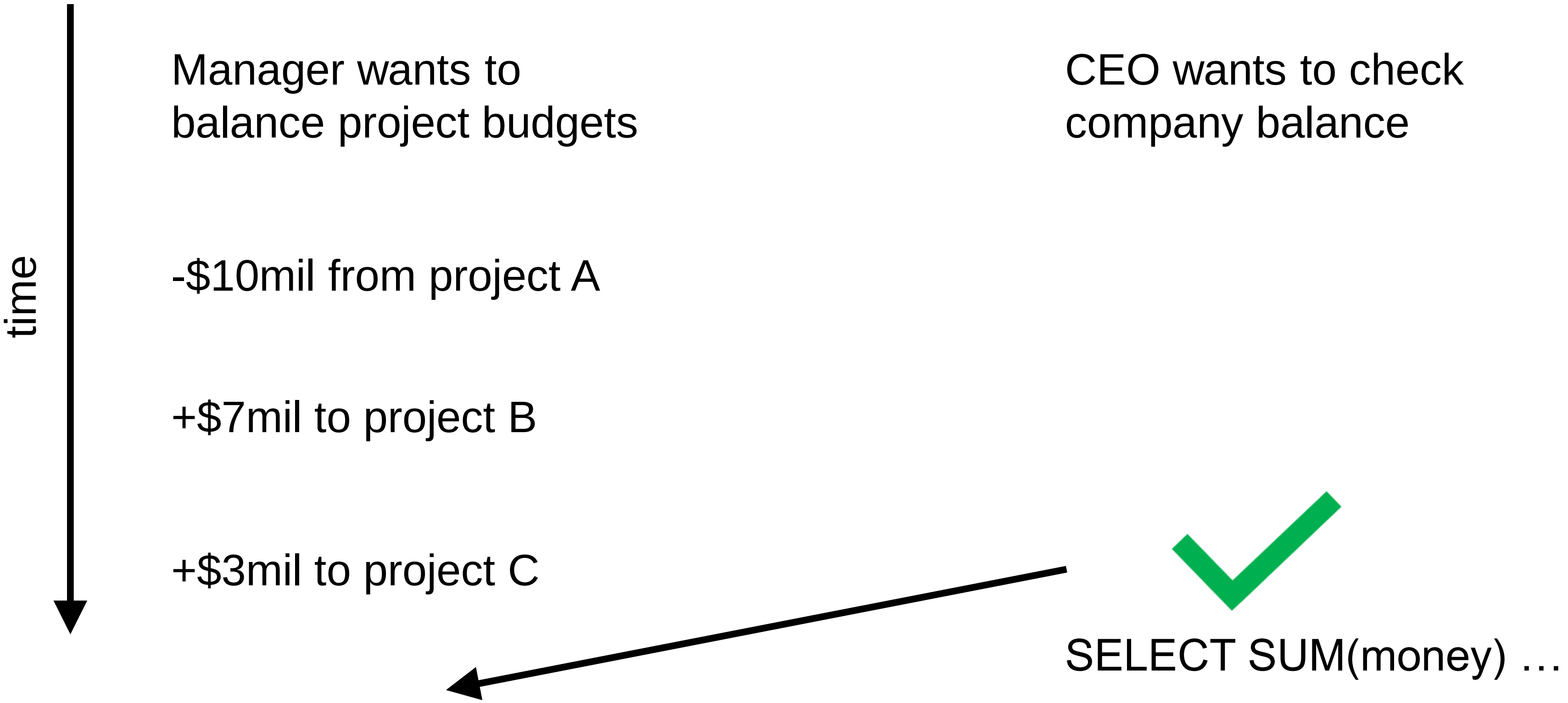
CEO wants to check company balance

SELECT SUM(money) …

✓

Dirty/Inconsistent Read

- Write-Read (WR) conflict
- Budget management scenario:



Manager wants to balance project budgets

CEO wants to check company balance

-$10mil from project A

+$7mil to project B

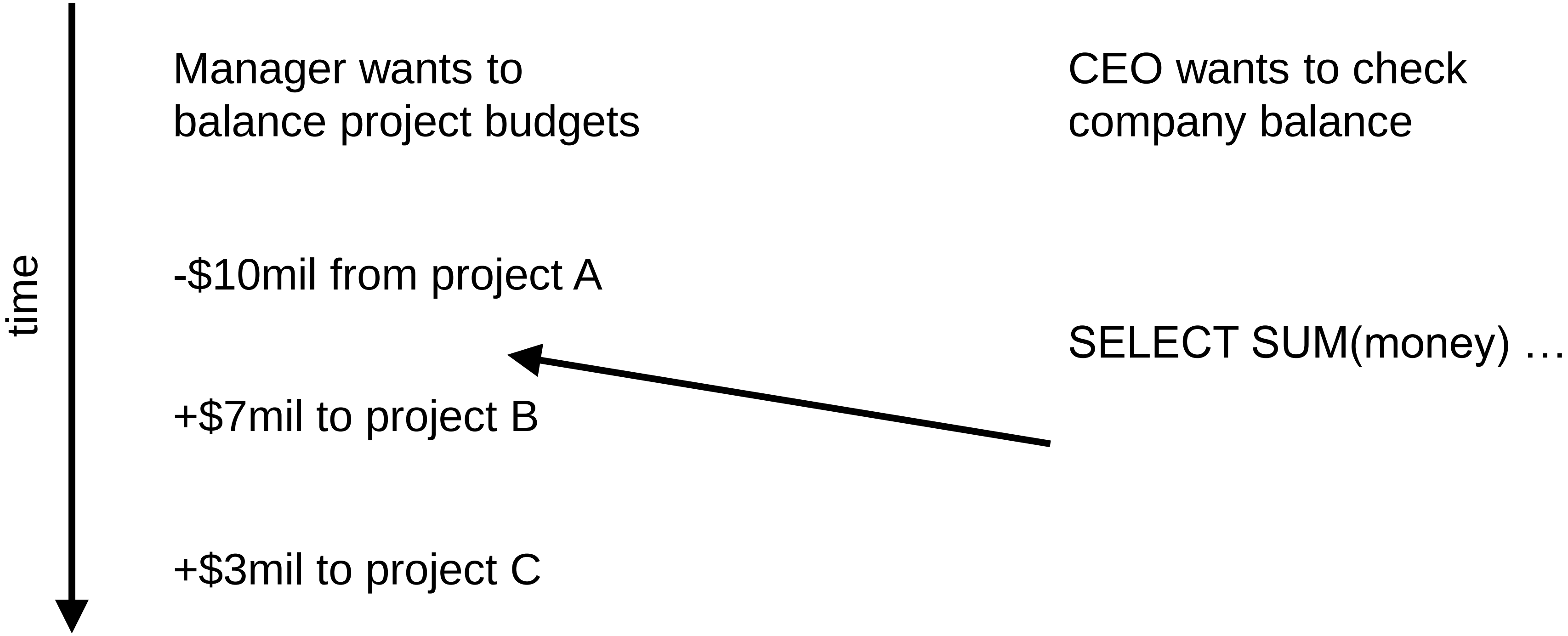
+$3mil to project C

✓

SELECT SUM(money) …

Dirty/Inconsistent Read

- Write-Read (WR) conflict
- Budget management scenario:



Manager wants to balance project budgets

CEO wants to check company balance

-$10mil from project A

SELECT SUM(money) …

+$7mil to project B

+$3mil to project C

Dirty/Inconsistent Read

- Write-Read (WR) conflict
- Budget management scenario:

time

Manager wants to balance project budgets

CEO wants to check company balance

-$10mil from project A

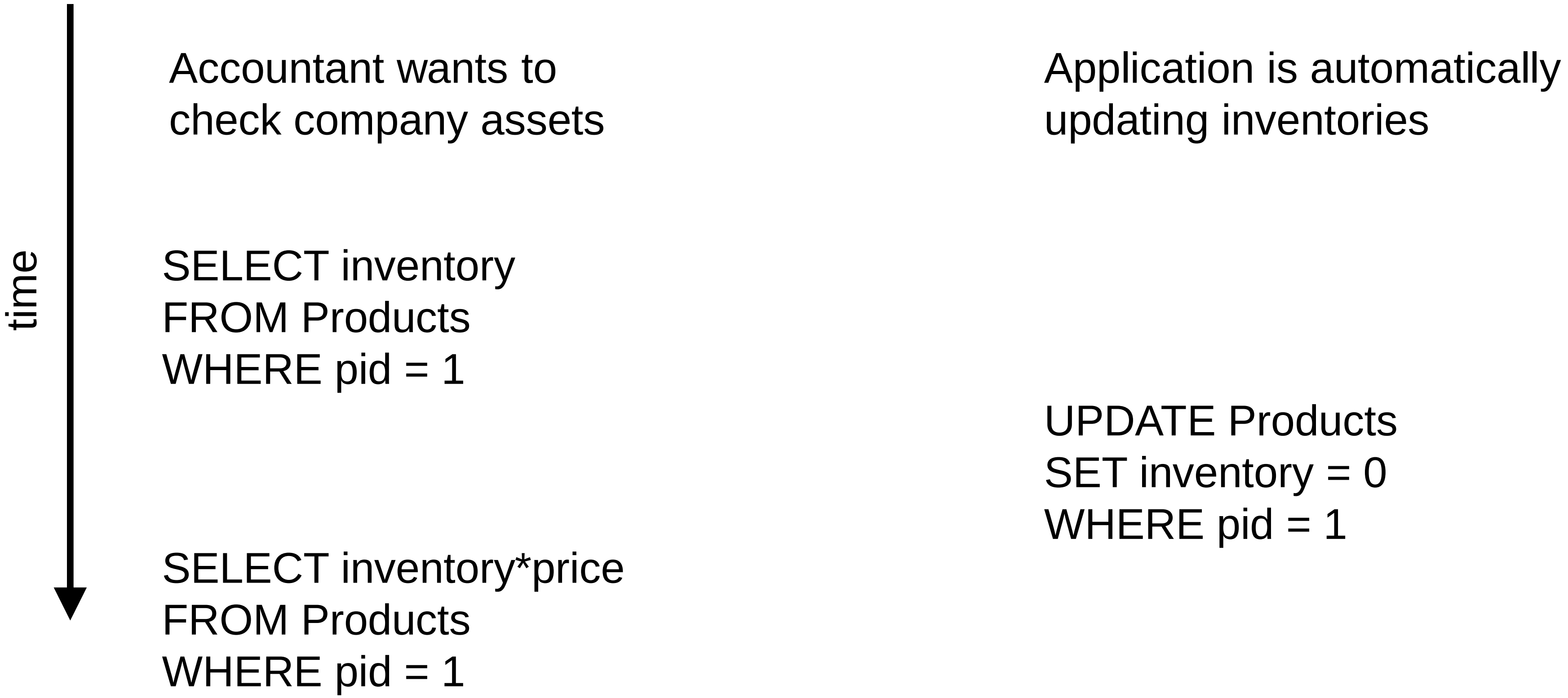
SELECT SUM(money) …

+$7mil to project B

+$3mil to project C

Unrepeatable Read

- Read-Write (RW) conflict
- Asset checking scenario:



Accountant wants to
check company assets

Application is automatically
updating inventories

```
SELECT inventory
FROM Products
WHERE pid = 1
```

```
UPDATE Products
SET inventory = 0
WHERE pid = 1
```

```
SELECT inventory*price
FROM Products
WHERE pid = 1
```

Unrepeatable Read

- Read-Write (RW) conflict
- Asset checking scenario:

time

Accountant wants to check company assets

Application is automatically updating inventories

```
SELECT inventory
FROM Products
WHERE pid = 1
```

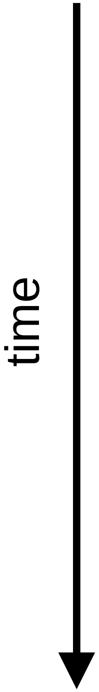
```
UPDATE Products
SET inventory = 0
WHERE pid = 1
```

```
SELECT inventory*price
FROM Products
WHERE pid = 1
```

Might get a value that doesn’t correspond to previous read!

Phantom Read

- Same read has more rows
- Asset checking scenario:



Accountant wants to check company assets

Warehouse catalogs new products

```
SELECT *
FROM products
WHERE price < 10.00
```

```
INSERT INTO Products
VALUES (‘nuts’, 10, 8.99)
```

```
SELECT *
FROM products
WHERE price < 20.00
```

Phantom Read

- Same read has more rows
- Asset checking scenario:

time

Accountant wants to check company assets

Warehouse catalogs new products

```
SELECT *
FROM products
WHERE price < 10.00
```

```
INSERT INTO Products
VALUES (‘nuts’, 10, 8.99)
```

```
SELECT *
FROM products
WHERE price < 20.00
```

Gets a row that should have been in the last read!

ACID

- Atomic
- Consistent
- Isolated
- Durable
- Ideally a DBMS follows these principles, but sacrificing good behavior for performance gains is common
- Definitely needs to follow these principles if you are dealing with $$$

ACID

- Atomic
- Consistent
- Isolated
- Durable
- Ideally a DBMS follows these principles, but sacrificing good behavior for performance gains is common
- Definitely needs to follow these principles if you are dealing with $$$

Atomic

- Operation encapsulation
- An operation is atomic if everything works or nothing happens

- 2nd most important in ACID! And feature we need for programming

Consistent

- Integrity constraints and application specification
- Operations assume a valid database state and end in a valid database state

Isolated

- Concurrency management
- Isolated behavior is as if an operation ran as if it was the only one running

- The most important feature of ACID.
- Atomicity + Isolation is what gives us functional transactions.

Durable

- Crash recovery
- CSE 444 topic
  - not discussed in this class (but very interesting!)

Transactions

- An application function may involve multiple different operations
- We want to make sure the parts of an operation execute properly **together as if it were a single action**
- We say that a transaction is one of these groups of executions
  - DBMS usually automatically treats each SQL statement as its own transaction unless otherwise specified

```
BEGIN TRANSACTION
 [SQL Statements]
COMMIT -- finalizes execution
```

```
BEGIN TRANSACTION
 [SQL Statements]
ROLLBACK -- undo everything
```

Concurrency Control Problems

- We’ve (sorta) solved the first problem!
- **DBMS concurrency control is all based on specification**
- Merely specifying what your transactions are is good enough for the DBMS to take care of it as a single unit

Transaction Modeling

- Logical perspective → a database is a set of sets/bags of tuples
- Design perspective → a database is a schema that models information
- Physical perspective → a database is a catalog of organized files
- Transaction perspective → a database is a **collection of elements** that can be **written to** or **read from**
  - Definition of element can vary depending on DBMS and/or user specification
  - Usually element = 1 block of database
  - Transactions are sequence of element reads and/or writes

Schedules

- Transactions are sequence of element reads and/or writes
  - $R_i(A)$ → **read** element A
  - $W_i(A)$ → **update** element A

  To add or remove tuples, we need more operations
  - $I_i(A)$ → **insert** an element A
  - $D_i(A)$ → **delete** an element A
- Schedules are a sequence of interleaved actions from all transactions

Serial Schedules

- A **serial schedule** is a schedule where each transaction is executed in some order, one after the other
- A **serializable schedule** is a schedule where transactions are executed with possible interleaving, but it appears to the outside as if they were executed in serial order
    - Given a before and after image of serializable compared to serial, you would not be able to tell if there was interleaving

Two Transactions

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

Example

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)

A **Serial** Schedule

T1	T2	A = 2
		B = 2

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

WRITE(B,t)		A = 102
		B = 102

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

	WRITE(B,s)	A = 204
		B = 204

A **Serial** Schedule

T1	T2	A = 2
		B = 2

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

	WRITE(B,s)	A = 4
		B = 4

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

WRITE(B,t)		A = 104
		B = 104

Serializable Schedule

A schedule is *serializable* if it is equivalent to a serial schedule

A **Serializable** Schedule

T1	T2	A = 2
		B = 2

READ(A, t)
t := t+100
WRITE(A, t)
		A = 102
		B = 2

	READ(A,s)
	s := s*2
	WRITE(A,s)
		A = 204
		B = 2

READ(B, t)
t := t+100
WRITE(B,t)
		A = 204
		B = 102

	READ(B,s)
	s := s*2
	WRITE(B,s)
		A = 204
		B = 204

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

A **Non-Serializable** Schedule

T1	T2	A = 2
		B = 2

READ(A, t)		

t := t+100		A = 102
		B = 2

WRITE(A, t)		
	READ(A,s)	

	s := s*2	A = 204
		B = 2

	WRITE(A,s)	
	READ(B,s)	

	s := s*2	A = 204
		B = 4

	WRITE(B,s)	
READ(B, t)		

t := t+100		A = 204
		B = 104

WRITE(B,t)		

A **Non-Serializable** Schedule

T1	T2	A = 2
		B = 2

READ(A, t)		A = 102
		B = 2

t := t+100		
WRITE(A, t)		

	READ(A,s)	A = 204
		B = 2

	s := s*2	
	WRITE(A,s)	

	READ(B,s)	A = 204
		B = 4

	s := s*2	
	WRITE(B,s)	
READ(B, t)		

t := t+100		A = 204
		B = 104

WRITE(B,t)		

Shouldn’t be possible!

Serial Schedule Example

- T1 then T2

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

time

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

Serial Schedule 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)	

Serializable Schedule

- 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)

Serializable Schedule

- 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)

Looks like T2 finished after T1 for each element

Serializable Schedule

- Not serializable to either order

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

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

Serializable Schedule

- Not serializable to either order

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

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

Looks like T2 finished last looking at A

Looks like T1 finished last looking at B

Checking Serializability

- How does the DBMS tell if some schedule is serializable?

- We define operation “**conflicts**” and check for their interactions in a schedule

Conflicts

- Most application concurrency problems are describable by conflicts
- Lost Update → Write-Write (WW) conflict
- Dirty Read → Write-Read (WR) conflict
- Unrepeatable Read → Read-Write (RW) conflict
- Phantom Read
  - We’ll talk about this later…

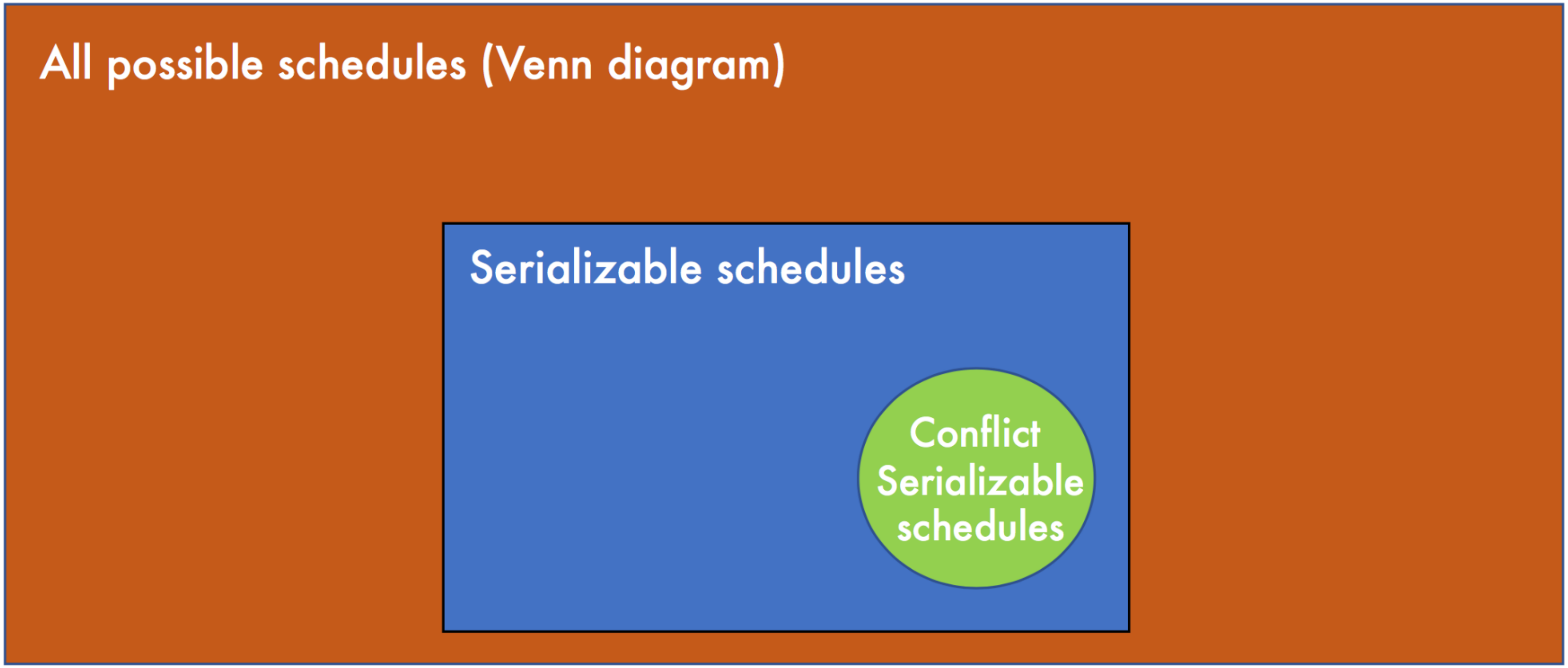
Individual conflicts aren’t “bad”!
Interleaving of conflicts can lead to trouble.

Types of Conflicts

- Changing the order of things in conflict will cause program behavior to behave badly
- **Intra-transaction conflicts**
  - Operations within a transaction cannot be swapped (you would be literally changing the program)
- **Inter-transaction conflicts**
  - WW conflicts → W1(X), W2(X)
  - WR conflicts → W1(X), R2(X)
  - RW conflicts → R1(X), W2(X)

Conflict Serializability

- Showing program serializability is hard
  - Needs lots of extra information besides R, W, I, D
- Observation: Enforce something something simpler but stronger than serializability


All possible schedules (Venn diagram)
Serializable schedules
Conflict Serializable schedules

Conflict Serializability

- Showing program serializability is hard
  - Needs lots of extra information besides R, W, I, D
- Observation: Enforce something something simpler but stronger than serializability
- **Conflict serializability implies serializability**
- Serializability does not imply conflict serializability

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)	

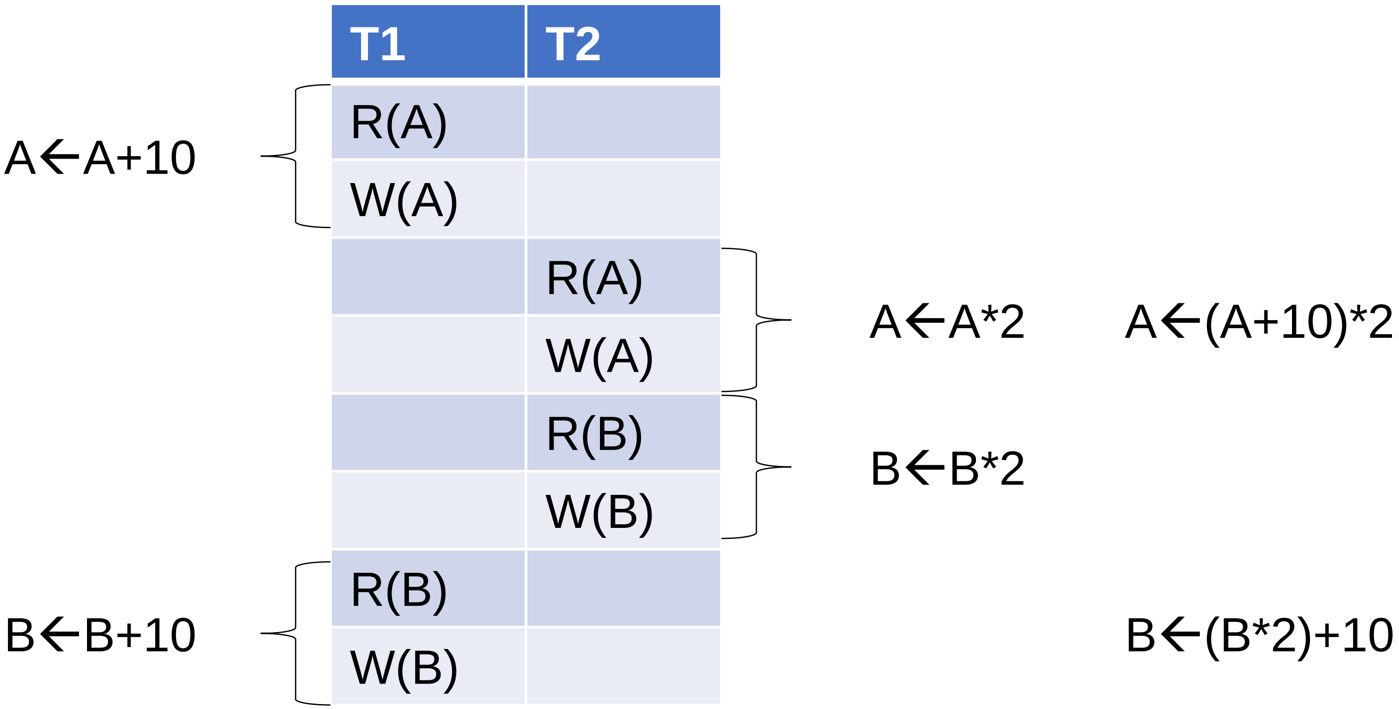
Non Conflict Serializable Schedule Example

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

Not serializable nor conflict serializable
(different results depending on if you add or multiply first)



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

A←A+10

A←A*2

A←(A+10)*2

B←B*2

B←B+10

B←(B*2)+10

Serializable vs Conflict Serializable

Serializable but not conflict serializable
(because the values ended up equivalent to serial schedule)

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

A←A+10

A←A+20   A←(A+10)+20

B←B+20

B←B+10

B←(B+20)+10

Enforcing Conflict Serializability

- We only care if some conflict rule would be broken (no need to micromanage)
- Need an effective algorithm

- Method:
  - Model each transaction as a node
  - Model a inter-transaction conflict as a directed edge
  - If the resulting graph is a DAG then there is a serial order
  - Conflict serializability enforcement turns into the graph cycle detection problem

Testing for Conflict-Serializability

Precedence graph:

- A node for each transaction T_i
- An edge from T_i to T_j whenever an action in T_i conflicts with, and comes before an action in T_j
- No edge for actions in the same transaction

Theorem:

The schedule is conflict-serializable iff the precedence graph is acyclic

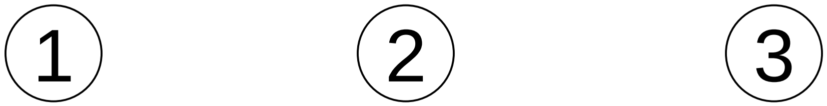
Testing for Conflict-Serializability

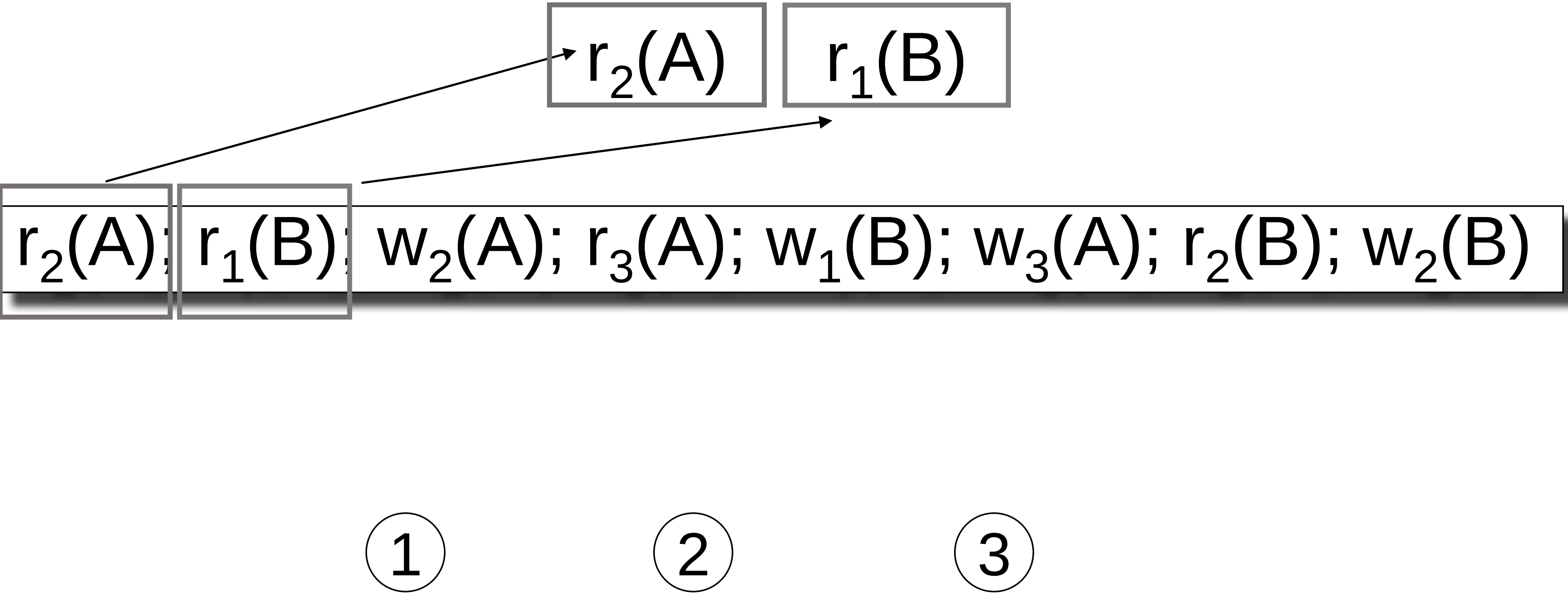
Important:

Always draw the full graph, unless ONLY asked if (yes or no) the schedule is conflict serializable

Example 1

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

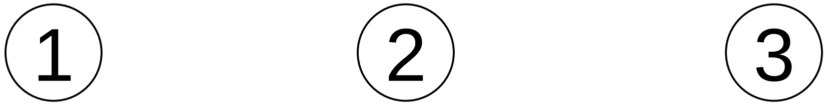




$r_2(A)$; $r_1(B)$; $w_2(A)$; $r_3(A)$; $w_1(B)$; $w_3(A)$; $r_2(B)$; $w_2(B)$

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

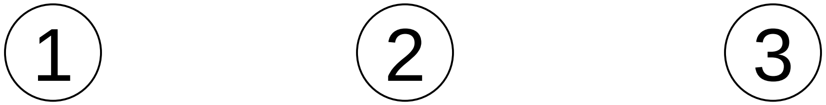
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



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

No edge because
no conflict (A != B)

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



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

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

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

No edge because same txn (2)

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

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

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

① ② ③

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

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

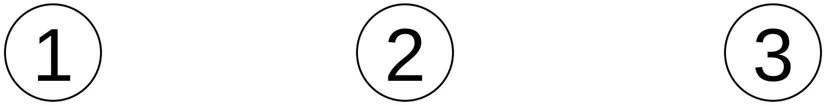
$\boxed{r_2(A)}$ $\boxed{w_3(A)}$?

$\boxed{r_2(A)}; r_1(B); w_2(A); r_3(A); w_1(B); \boxed{w_3(A)}; r_2(B); w_2(B)$

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

Edge! Conflict from T2 to T3

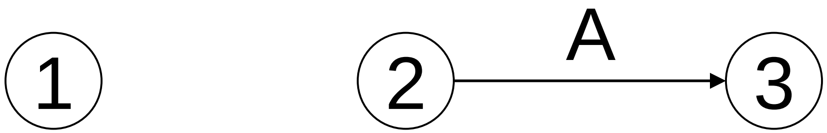
$r_2(A)$; $r_1(B)$; $w_2(A)$; $r_3(A)$; $w_1(B)$; $w_3(A)$; $r_2(B)$; $w_2(B)$



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

Edge! Conflict from
T2 to T3

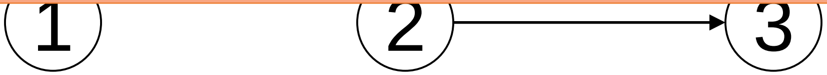
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



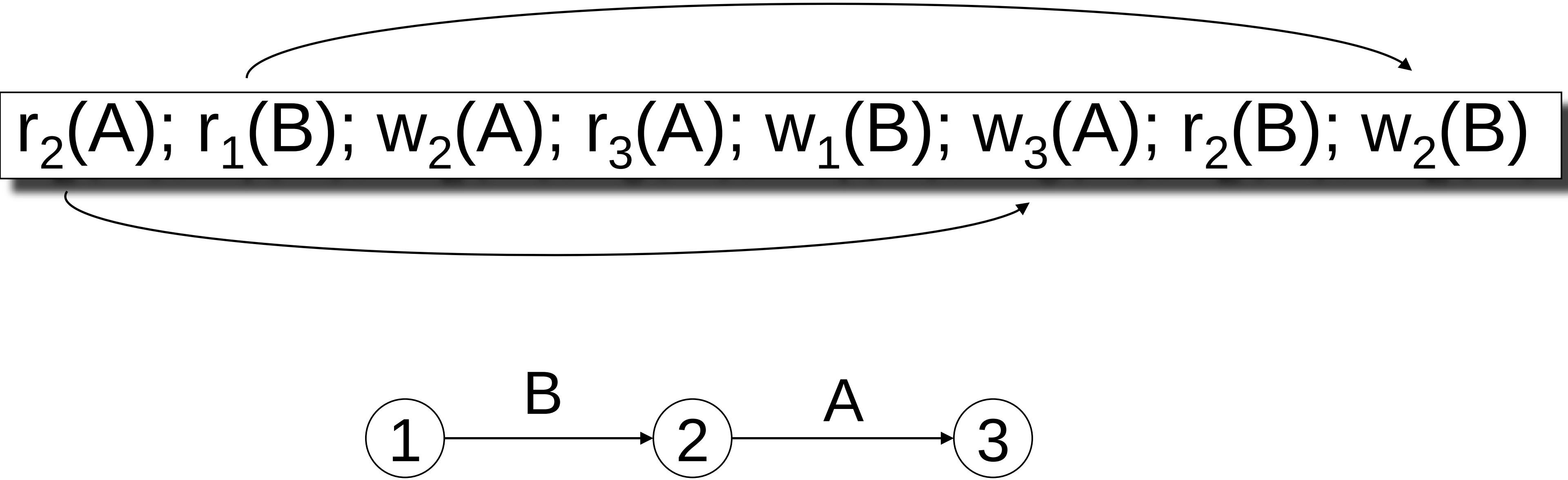
$\boxed{r_2(A)}$ $\boxed{r_2(B)}$?

$\boxed{r_2(A)}$; $r_1(B)$; $w_2(A)$; $r_3(A)$; $w_1(B)$; $w_3(A)$; $\boxed{r_2(B)}$; $w_2(B)$

And so on until compared every pair of actions…



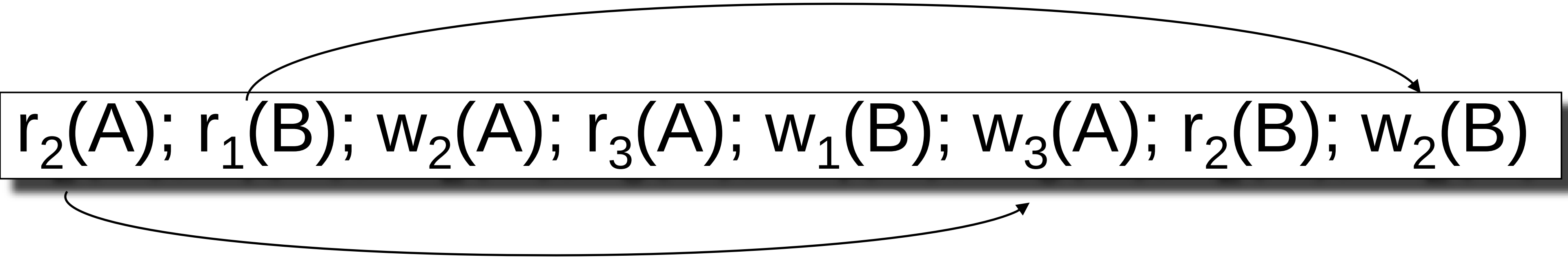
Example 1



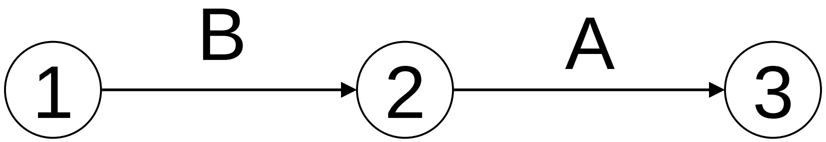
$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$

More edges, but repeats of the same directed edge not necessary

Example 1



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

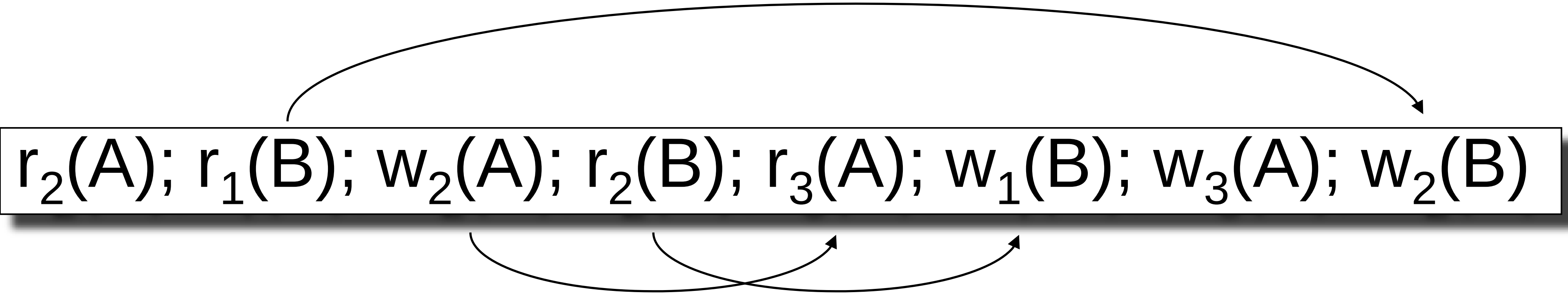


This schedule is conflict-serializable

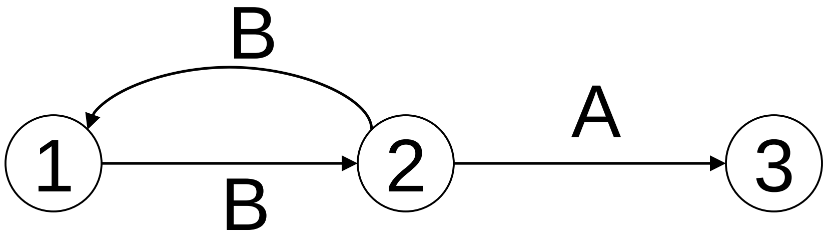
Example 2

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

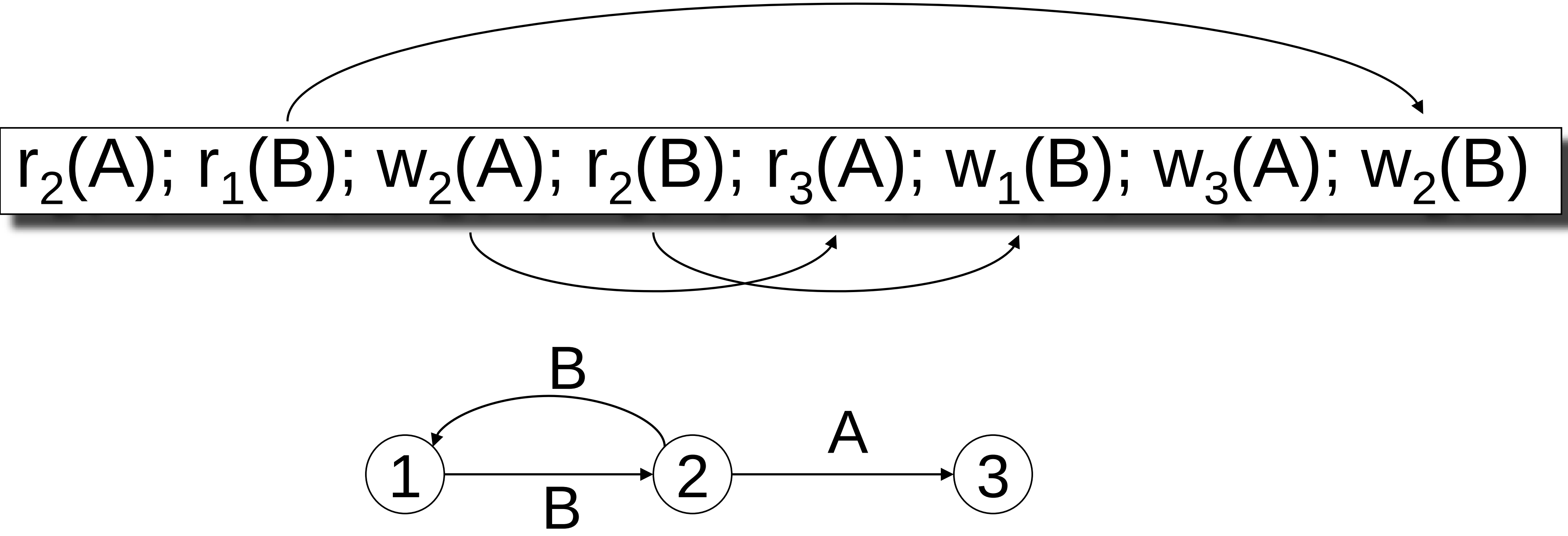
Example 2



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



Example 2



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

This schedule is NOT conflict-serializable